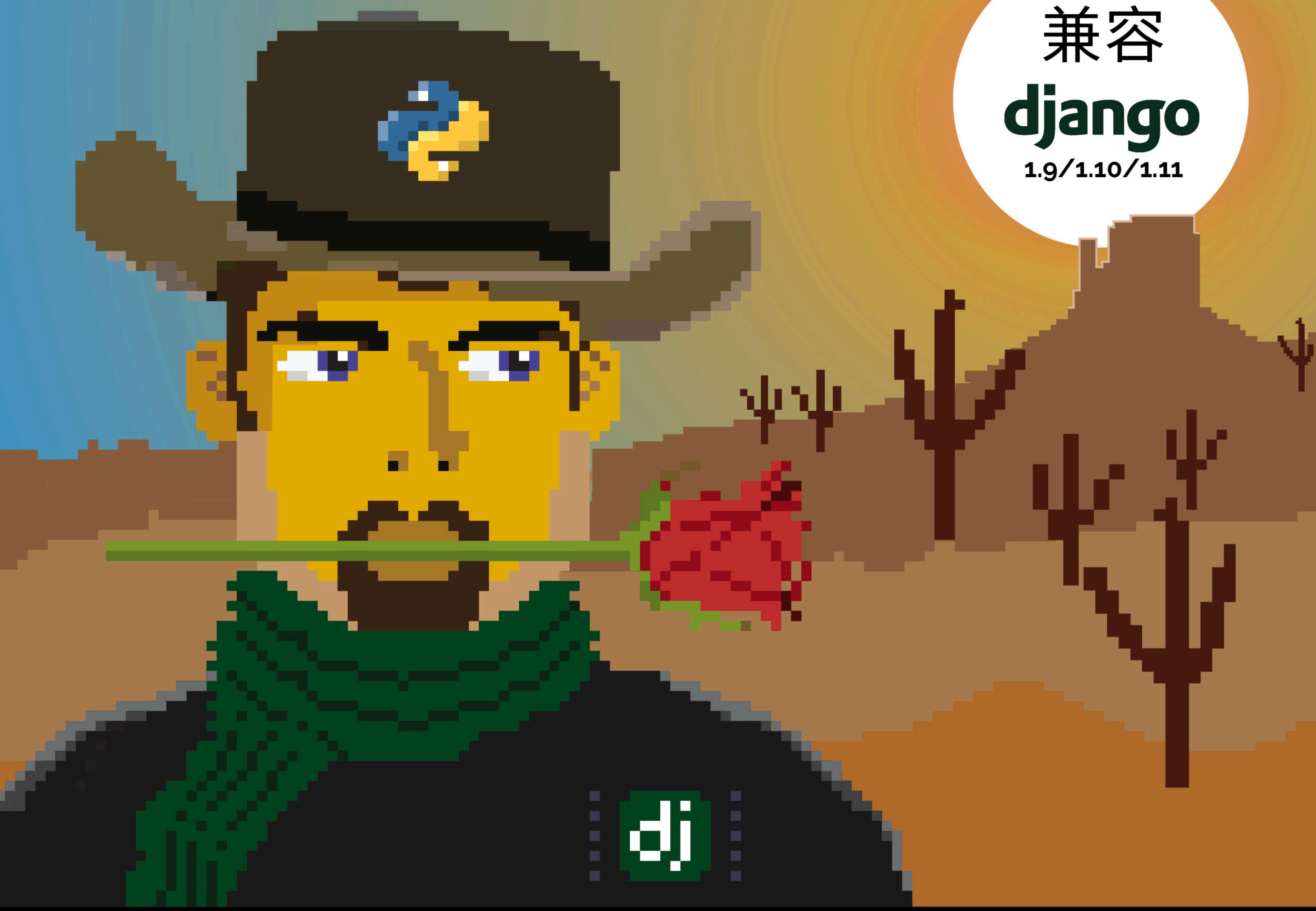
Leif Azzopardi & David Maxwell 著 / 安道 译

Django 基础教程

Tango with Django

兼容 django 1.9/1.10/1.11



www.tangowithdjango-china.com

Django 基础教程

Tango with Django

Leif Azzopardi & David Maxwell 著

安道 译

rev 0.0.1, 2018-03-12T20:09:04+08:00

❍ 目录

本书是《Tango with Django》的简体中文版，由 Leif Azzopardi 和 David Maxwell 授权翻译。

1 导言

这一本学做结合的指南，旨在教你使用 Django 和 Python 做 Web 开发。本书主要针对学生，因此会详解使用 Django 开发 Web 应用过程中的每个步骤。

Django 官方提供了一份教程，而且网上也有很多优秀的教程，本书的目标是填补一些空白，通过实例开发学习 Django 框架。此外，本书还会介绍开发 Web 应用所需掌握的其他知识，例如 HTML、CSS、JavaScript 等等。

1.1 本书特色

- 事半功倍

笔者见过很多聪明的学生陷入僵局，浪费几小时的时间尝试解决遇到的 Django 或其他 Web 开发问题。这些问题往往是由于抓不住重点，或者所用的材料言语不详。有时，你可能灵光一现，在十几分钟之内解决问题，但是更多的时候要耗费几小时。笔者结合自己的经验，力求消除这些障碍，让你远离坎坷，在开发应用的过程中一帆风顺。

- 平缓学习曲线

Web 应用框架省时省力，但前提是你知道怎么使用框架。框架的学习曲线往往陡峭。本书力求平缓学习曲线，详解方方面面，让你快速掌握框架的用法。

- 改进工作流程

使用 Web 应用框架要遵守特定的设计模式，你只需在特定的位置放置特定的代码。但是，与很多学生交流之后，笔者发现他们经常抱怨，不知如何从框架那里夺回控制权（即控制反转）。为此，笔者制定了几个工作流程，帮你在开发过程中重获控制权，以自己的方式构建 Web 应用。

- 边学边做

无论如何，不要只是看看内容而已。这是一本实作指南，你要自己动手使用 Django 构建 Web 应用。动眼不动手可不行！若想有所获益，请跟着本书一起开发应用。而且，在开发的过程中，不

要复制粘贴书中的代码。自己动手输入，想想代码的作用，然后再阅读书中给出的说明。如果依旧不解，查阅 Django 文档，到 Stack Overflow 或其他网站中寻求帮助，一定要把不理解的地方弄明白。如果你确实陷入僵局了，可以联系笔者，以便笔者改进内容——已经有多位读者为本书做出了贡献。

1.2 你将学到

本书以一个示例为主线，说明如何开发一个名为 Rango 的 Web 应用。在这个过程中将讲解如何完成下述关键任务：

- **搭建开发环境**，包含学习使用终端、虚拟环境、pip 安装程序和 Git，等等。
- **创建 Django** 项目和简单的 Django 应用。
- **配置 Django** 项目，伺服静态媒体和其他媒体文件。
- 使用 Django 的“**模型-视图-模板**”设计模式。
- **创建数据库模型**，使用 Django 提供的对象关系映射（Object Relational Mapping，ORM）功能。
- **创建表单**，利用数据库模型生成动态网页。
- 使用 Django 提供的**用户身份验证**服务。
- 在 Django 应用中融合**外部服务**。
- 在 Web 应用中引入层叠样式表（Cascading Styling Sheet，CSS）和 JavaScript。
- 使用 CSS 为应用提供专业的外观。
- 在 Django 应用中处理 **cookie** 和**会话**。
- 在应用中使用 Ajax 等高级技术。
- 把应用**部署**到 PythonAnywhere 上。

每一章结尾都有几道练习题，旨在加强你对知识的掌握，也检验你能不能学以致用。后面几章还有开放性开发练习，而且提供了参考代码和说明。

</> 这样的区域是练习

每一章都有练习题，旨在检查你对知识和技能的掌握情况。你必须解答这些练习，因为后面的章节建立在这些题目之上。

别担心自己做不出来，本书的 GitHub 仓库中有所有练习题的解答。

1.3 用到的技术和服务

本书将使用的技术和外部服务如下：

- Python 编程语言
- pip 包管理器
- Django 框架
- Git 版本控制器系统
- GitHub
- HTML
- CSS
- JavaScript 编程语言
- jQuery 库
- Twitter Bootstrap 框架
- Webhose API（即后文所用的搜索 API）
- PythonAnywhere 托管服务

这些技术和服务是笔者精选的，其中某些是 Web 开发的基础。Twitter Bootstrap 为 Web 应用提供样式，Webhose API 是一个外部服务，PythonAnywhere 则能简化部署应用的过程。

1.4 Rango 的初步设计和客户要求

本书的主线是开发一个名为 Rango 的应用。在这个过程中，我们将涵盖构建 Web 应用所需掌握的重要知识。如果想查看这个应用的最终版本，请访问 *http://rangodemo.pythonanywhere.com/rango/*。

设计概要

客户要求你开发一个名为 Rango 的网站，让用户按分类浏览不同的网页。在西班牙语中，“rango”的意思是“等级”或“显要地位”。

- 用户访问 Rango 网站的首页时，客户想让访客看到：

- 访问次数最多的 5 个网页
- 查看次数最多的 5 个分类
- 浏览或搜索分类的不同方式

❑ 用户访问分类页面时，客户想让 Rango 显示：

- 分类的名称、查看次数、点赞次数，以及分类下的网页（显示网页的标题，并链接到网页的 URL）
- 一定的搜索功能（通过搜索 API 实现），找出可以归入当前分类的网页

❑ 客户要求记录分类的名称，分类页面被查看的次数，以及多少用户点击了"点赞"按钮（即用户对分类打分，投票排名）。

❑ 各分类能通过友好的 URL 访问，例如 */rango/books-about-django/*。

❑ 只有注册用户能搜索，能把网页添加到分类中。因此要让网站的访客能注册账户。

这么一看，我们要开发的应用并不复杂，不过是列出一些网页的分类而已。然而，这其中有些复杂问题需要解决，可能并不像想的那样简单。着手开发之前，我们先规划一下整体设计。

</> 练习

继续阅读之前，请根据客户要求绘制下述设计图：

❑ N 层或系统架构图

❑ 主页和分类页面的线框图

❑ 应用的 URL 映射

❑ 实体关系图（Entity-Relationship diagram，ER 图），描述要实现的数据模式

在继续阅读之前，请试着完成这几题。即使你不熟悉系统架构图、线框图或 ER 图也没关系，这几题的重点是让你思考如何说明和描述即将构建的应用。

N 层架构

多数 Web 应用的整体架构是一种 3 层结构。Rango 也采用这种架构，不过稍有不同，因为它还要

与外部服务交互。

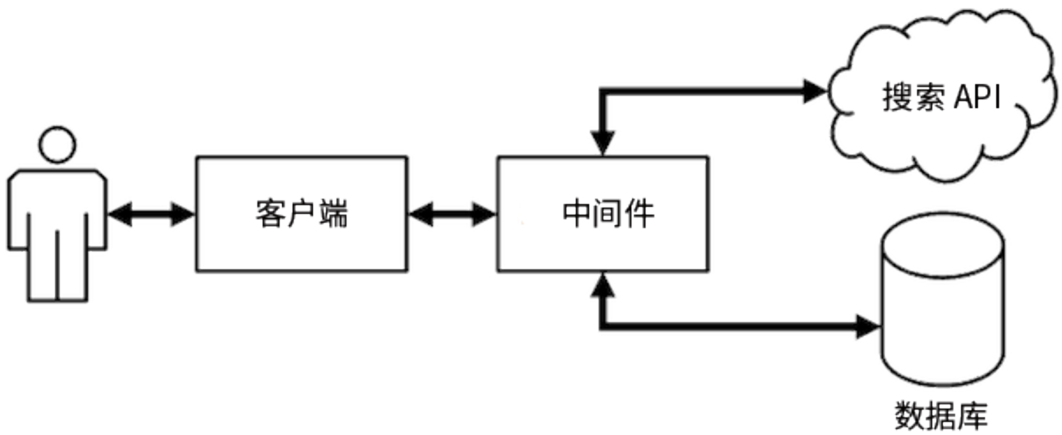


图 1-1：Rango 的 3 层系统架构

因为我们将使用 Django 构建这个 Web 应用，所以各层用到的技术如下：

- 客户端是 Web 浏览器（例如 Chrome、Firefox 或 Safari），负责渲染 HTML/CSS 页面。
- 中间件是一个 Django 应用程序，在开发过程中由 Django 内置的开发 Web 服务器负责调度。
- 数据库使用基于 Python 的 SQLite3 数据库引擎。
- 搜索 API 使用 Webhose API。

本书基本上将集中精力开发中间件，不过从图 1-1 中不难看出，我们也将与其他组件交互。

线框图

线框图可以让客户一览最终完成的应用是什么样子。线框图能节省大量时间，不过具体形式各种各样，有手绘的，也有使用专用工具的。Rango 应用首页的设计稿见图 1-2，分类页面的设计稿见图 1-3。

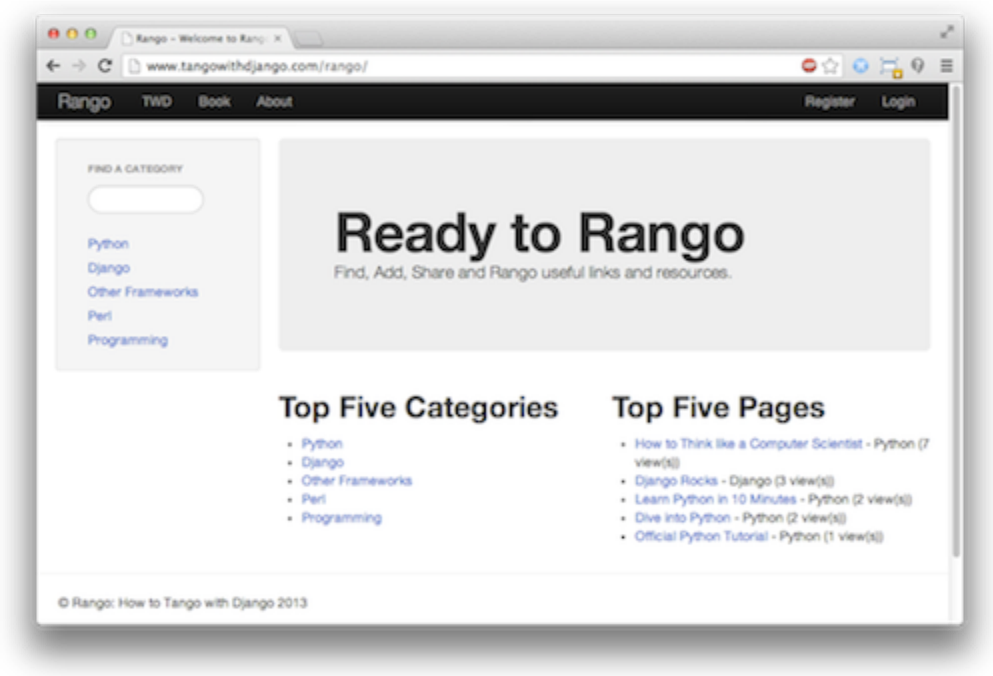


图 1-2：首页，左边是分类搜索框，右边是查看次数最多的 5 个分类和 5 个网页

页面和 URL 映射

从客户的要求中我们知道，应用至少要有两个页面。为了能访问各个页面，我们要描述 URL 映射。你可以把 URL 映射理解为访问页面时在浏览器地址栏中输入的文本。Rango 应用的基本 URL 映射如下：

- / 或 */rango/* 指向主页
- */rango/about/* 指向关于页面
- */rango/category/<category_name>/* 指向名为 <category_name>（例如 games、python-recipes 或 code-and-compilers）的分类页面

这只是开始，在构建应用的过程中，可能还要添加其他 URL 映射。本书将带领你使用 Django 框架和“模型-视图-模板”设计模式逐渐丰富这些页面的内容。对 URL 映射和页面的设计有了大致了解之后，我们要定义数据模型，存储应用将用到的数据。

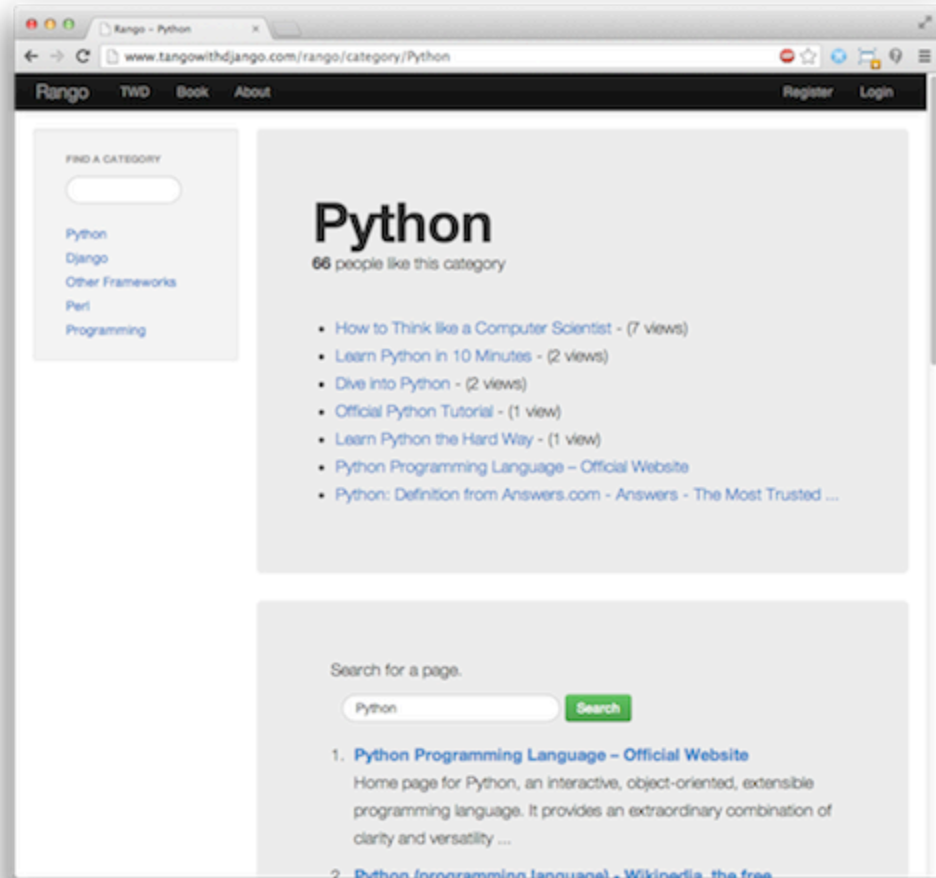


图 1-3：分类页面，显示分类中的网页（带有访问次数），以及搜索“Python”得到的结果

实体关系图

根据客户的要求，很明显我们至少需要两个实体：分类（category）和网页（page）。而且，一个分类中可以有多个网页。这个简单的数据模型可以通过下述 ER 图描述。

注意，这个关系并不十分明确。理论上，一个网页可以归入多个分类。鉴于此，分类和网页之间可以通过多对多关系建模。但是这样处理太过复杂，因此我们将假定一个分类中有多个网页，而一个网页只能属于一个分类。这样并不妨碍把一个网页归入不同的分类，只不过要多次输入相同的网页，不是太理想而已。



图 1-4：Rango 应用中两个主要实体的 ER 图

■ 做好记录 ■

要养成习惯，把所做的假设记录下来，例如这里假设的一对多关系，以防以后出问题。有了记录，你便可以与开发团队沟通，确保他们考虑到了你的假设。

根据这一假设，我们可以通过表格列出各实体的详细内容，指明各实体中有哪些字段。我们使用 Django 的 `ModelField` 类型定义各字段的类型（即 `IntegerField`、`CharField`、`URLField` 或 `ForeignKey`）。注意，在 Django 中主键（primary key）是隐含的，Django 会为各模型添加 `id` 字段（详情参见第 5 章）。

表 1-1：Category 模型

字段	类型
`name`	`CharField`
`views`	`IntegerField`
`likes`	`IntegerField`

表 1-2：Page 模型

字段	类型
`category`	`ForeignKey`
`title`	`CharField`
`url`	`URLField`
`views`	`IntegerField`

此外，还需要一个 `User` 模型，以便用户注册和登录。这里没有给出 `User` 模型，讨论用户身份验证时再介绍。后面的章节将说明如何在 Django 中定义这些模型，以及如何使用内置的 ORM 连接数据库。

1.5 小结

这些整体上的设计和要求将在构建 Web 应用的过程中为我们提供参考。虽然我们将集中精力说明如何使用特定的技术，但是多数数据库驱动的网站都是这么规划的。因此，你最好能熟练地阅读和制定这样的要求和设计，以便与人顺畅沟通。本书将使用 Django 和相关的技术实现这里制定的

要求。

◆ 复制粘贴代码 ◆

阅读本书的过程中，你可能会把书中的代码复制粘贴到代码编辑器中。然而，笔者建议自己动手输入代码。我们知道自己输入稍显麻烦，但是这样有助于你理解整个过程，还能让你记住以后用得到的命令。

此外，复制粘贴 Python 代码简直就是自找麻烦。复制的空白可能变成空格、制表符或二者混杂。这样可能导致各种奇怪的问题，而且不一定是由缩进引起的。如果你确实想复制粘贴代码，一定要留神这样的问题。如果你使用的是 Python 3，对这样的问题要格外小心，因为混用制表符和空格缩进会导致 `TabError`。

多数代码编辑器都能显示空白，而且会区分制表符和空格。如果你使用的编辑器有这样的功能，一定要启用，免得分不清。

2 前期准备工作

开始编程之前，我们要搭建好 Django 所需的开发环境。我们要在自己的电脑中安装所需的全部组件。本章概述要安装和使用的 5 个关键组件。

- 终端或命令提示符
- Python
- Python 包管理器和虚拟环境
- 集成开发环境（如果你想用的话）
- 版本控制系统 Git

如果你的电脑中已经安装了 Python 2.7/3.4/3.5 和 Django 1.9/1.10，而且熟悉前述工具，可以直接跳到第 3 章。否则，请阅读下面对这些组件的介绍，以及它们在开发过程中的重要性。我们还将指出这些组件的安装方式。

★ 开发环境 ★

搭建开发环境是个繁琐的过程，很容易出错，但不是每天都要做。本章介绍这其中涉及的关键技术，以及如何安装各个组件。

根据笔者的经验，建议你把搭建的步骤记录下来。说不定有一天你会用到，例如你买了新电脑，或者帮助别人。现在做的记录能为以后节省时间和精力。不要只看眼前！

2.1 Python

本书要求你在自己的电脑中安装 Python 编程语言，2.7 系列（至少 2.7.5）或 3.4 版本以上都行。

如果你不知道如何安装 Python，需要帮助，请阅读 A.1 节。

★ 不会用 Python? ★

如果你没用过 Python，或者想提升自己的技能，笔者推荐下面几个材料：

- Stavros 写的 Python 10 分钟速成教程
- Python 官方教程
- Allen B. Downey 写的《像计算机科学家一样思考 Python》
- Jennifer Campbell 和 Paul Gries 开设的“学习编程”课程

这几个材料能让你熟悉 Python 的基础知识，以便开始使用 Django 开发应用。注意，为了使用 Django，你无需变身专家。Python 是门优秀的语言，如果你会用其他编程语言，可以边用边学。

2.2 Python 包管理器

pip 是 Python 的包管理器。你可以使用 pip 安装各种库，增强 Python 的功能。

包管理器，不管是针对 Python 的、针对操作系统的，还是针对其他环境的，是一种自动安装、升级、配置和删除包的软件，解放了你的双手，无需你自己动手下载、安装和维护软件。Python 包管理起来可不简单。多数包通常有依赖（dependency），要随包一起安装。因此，包之间可能有冲突，需要依赖特定的版本。此外，包的系统路径也要指定和维护。幸好，这些繁琐的工作都由 pip 代为处理了，解放了我们的生产力。

使用 `pip` 命令运行 pip 试试看。如果提示找不到 `pip` 命令，说明你要安装 pip。详情参见附录 A。你还要确保自己的系统中安装了下面两个包。执行下述命令，安装 Django 和 Pillow（处理图像的 Python 库）：

```
$ pip install -U django==1.9.10
$ pip install pillow
```

■ 无法成功安装 Pillow? ■

安装 Pillow 时可能会报错，提示缺少 JPEG 支持，如下所示：

```
ValueError: jpeg is required unless explicitly disabled using
            --disable-jpeg, aborting
```

如果你遇到这个问题，禁用 JPEG 支持试试：

```
$ pip install pillow --global-option="build_ext"
                     --global-option="--disable-jpeg"
```

这样虽然无法处理 JPEG 图像，但是却能成功安装 Pillow。本书只要求能安装 Pillow 就行了。详情参见 Pillow 文档。

2.3 虚拟环境

环境就快搭建好了。但是在继续之前要注意一点，目前的环境虽然可以使用了，但是有些缺点。如果另一个应用要使用 Python 的其他版本才能运行怎么办？如果你想把 Django 换成最新版，但是依然想维护使用 Django 1.9 开发的项目又怎么办？？

答案是使用虚拟环境。借助虚拟环境，我们可以让不同的 Python 版本和同一个包的不同版本和谐相处。如今，这是人们普遍使用的 Python 环境搭建方式。

虽然不必须使用虚拟环境，但是强烈建议你使用。搭建、创建和使用虚拟环境的详细说明参见 A.4 节。

2.4 集成开发环境

集成开发环境（Integrated Development Environment，IDE）虽然不是必须的，但一个支持 Python 的好 IDE 能为开发工作提供极大的帮助。支持 Python 的 IDE 很多，JetBrains 出品的 PyCharm 和 PyDev（Eclipse 的插件）或许是其中最受欢迎的。Python Wiki 中有支持 Python 的 IDE 的最新列表。

自己研究一下，找出合用的。注意，有些 IDE 需要购买许可证。如果你选择的 IDE 能集成 Django，那就更好了。

笔者使用的是 PyCharm，因为它支持虚拟环境，而且能集成 Django（不过需要配置）。这里不讨论如何集成 Django，如果你想知道，可以阅读 JetBrains 网站中介绍 PyCharm 集成 Django 的文章。

2.5 代码仓库

最后，开发过程中应该把代码纳入版本控制系统，例如 SVN 或 Git。为了不脱离主线，这里不说明如何使用版本控制器系统。若想快速掌握 Git，推荐阅读《GitHub入门与实践》一书。强烈建议你为自己的项目创建一个 Git 仓库。

</> 练习

为了熟悉搭建环境的步骤，请尝试完成以下练习：

- 安装 Python 2.7.5+/3.4+ 和 pip
- 熟悉命令行界面，创建一个名为 *workspace* 的目录，我们创建的项目将保存在这里
- 搭建虚拟环境（选做）
- 安装 Django 和 Pillow
- 如果没有 Git 仓库托管网站（例如 GitHub、BitBucket，等等）的账号，注册一个
- 下载并设置一个 IDE，例如 PyCharm

前面说过，本书的内容和示例应用在一个 GitHub 仓库中。

- 如果你发现错误或其他问题，请在 GitHub 中提出，让我们知道
- 如果碰到做不出的练习，可以参照仓库中的代码

★ 目录是什么？ ★

前面的练习中有一题让你创建一个目录（directory），可目录究竟是什么呢？如果你一直使用 Windows 系统，目录就是文件夹（folder）。二者在概念上是一样的，都是一种目录结构，列出里面的文件和目录。

3 Django 基础

下面开始学习 Django。本章介绍如何新建项目和 Web 应用，最终让一个 Django 驱动的简单网站运行起来。

3.1 检查环境

首先，我们要检查有没有正确安装 Python 和 Django。请打开一个新终端窗口，执行下述命令，查看 Python 的版本。

```
$ python --version
```

得到的结果应该是 2.7.11 或 3.5.1，不过 2.7.5+ 或 3.4+ 版的 Python 也可以。如果需要安装或升级 Python，请翻到附录 A。

如果你想使用虚拟环境，记得激活。倘若不记得怎么操作了，请翻到 A.4 节。

确认 Python 正确安装之后，还要检查 Django。在终端窗口中执行下述命令，运行 Python 解释器。

```
$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在提示符后输入下述命令：

```
>>> import django
>>> django.get_version()
'1.9.10'
```

```
>>> exit()
```

如果一切正常，应该能看到 Django 的版本号。然后输入 `exit()`，退出 Python 解释器。如果无法导入 Django，确认你是不是在虚拟环境中， 再使用 `pip list` 命令查看安装的包。

如果不知如何安装包，或者安装的版本不对，请翻到附录 A，或者阅读 Django 文档中的安装说明。

■ 提示符 ■

本书代码片段中有两种符号要注意。

以美元符号（`$`）开头的代码片段，其后的内容是要在终端或命令提示符中运行的命令。

`>>>` 表示后面的命令要输入交互式 Python 解释器。解释器打开的方法是执行 `python` 命令，退出方法是输入 `exit()`。

3.2 创建 Django 项目

进入 *workspace* 目录，执行下述命令，新建一个 Django 项目：

```
$ django-admin.py startproject tango_with_django_project
```

如果你的电脑中没有 *workspace* 目录，那就创建一个，把 Django 项目和其他项目都保存在那里。我们将在代码中使用 `<workspace>` 指代你的 *workspace* 目录。你要把 `<workspace>` 替换成 *workspace* 目录的具体路径，例如 */Users/leifos/Code/* 或 */Users/maxwelld90/Workspace/*。

★ 找不到 `django-admin.py`？ ★

输入 `django-admin` 试试。根据采用的安装方式，有些系统可能无法识别 `django-admin.py`。

根据 Stack Overflow 中这个问答的建议，在 Windows 系统中可能要使用 *django-admin.py* 脚本的完整路径，例如：

```
python c:\python27\scripts\django-admin.py
       startproject tango_with_django_project
```

这个命令调用 *django-admin.py* 脚本，新建一个名为 *tango_with_django_project* 的 Django 项目。通常，笔者喜欢在 Django 项目所在的目录名后面加上 `_project`，明确表明目录中是什么。不过，具体怎么命名完全由你自己决定。

此时你会发现，你的 *workspace* 目录中出现了与项目同名的一个目录，即 *tango_with_django_project*。在这个目录中你会看到两个内容：

- 另一个与项目同名的目录
- 一个 Python 脚本，名为 *manage.py*

在本书中，我们将把内部那个 *tango_with_django_project* 目录称为项目配置目录。在这个目录中，你会看到 4 个 Python 脚本，下面简单介绍一下，后文再详细说明：

- *__init__.py*：一个空 Python 脚本，存在的目的是告诉 Python 解释器，这个目录是一个 Python 包；
- *settings.py*：存放 Django 项目的所有设置；
- *urls.py*：存放项目的 URL 模式；
- *wsgi.py*：用于运行开发服务器和把项目部署到生产环境的一个 Python 脚本。

项目目录中有个名为 *manage.py* 的文件，在开发过程中时常用到。它提供了一系列维护 Django 项目的命令，例如通过它可以运行内置的 Django 开发服务器，可以测试应用，还可以运行多个数据库命令。几乎每个 Django 命令都要调用这个脚本。

★ Django 管理脚本 ★

Django 管理脚本的详细说明参见 Django 文档。

执行 `python manage.py help` 命令可以查看可用命令列表。

你现在就可以使用 *manage.py* 脚本，执行下述命令试试：

```
$ python manage.py runserver
```

这个命令启动 Python，让 Django 运行内置的轻量级开发服务器。你在终端窗口中应该会看到类似下面的输出：

```
$ python manage.py runserver
```

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may
not work properly until they are applied.

Run 'python manage.py migrate' to apply them.

October 2, 2016 - 21:45:32
Django version 1.9.10, using settings 'tango_with_django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

从这段输出可以看出几件事。首先，没有出现阻碍应用运行的问题。但是，输出中有个提醒，指出有未应用的迁移。这个问题在设置数据库时再讨论，现在暂且忽略。最后，尤为重要的是一个 URL 地址，即 *http://127.0.0.1:8000/*，这是 Django 开发服务器的地址。

打开 Web 浏览器，输入 URL *http://127.0.0.1:8000/*。你将看到类似图 3-1 的网页。

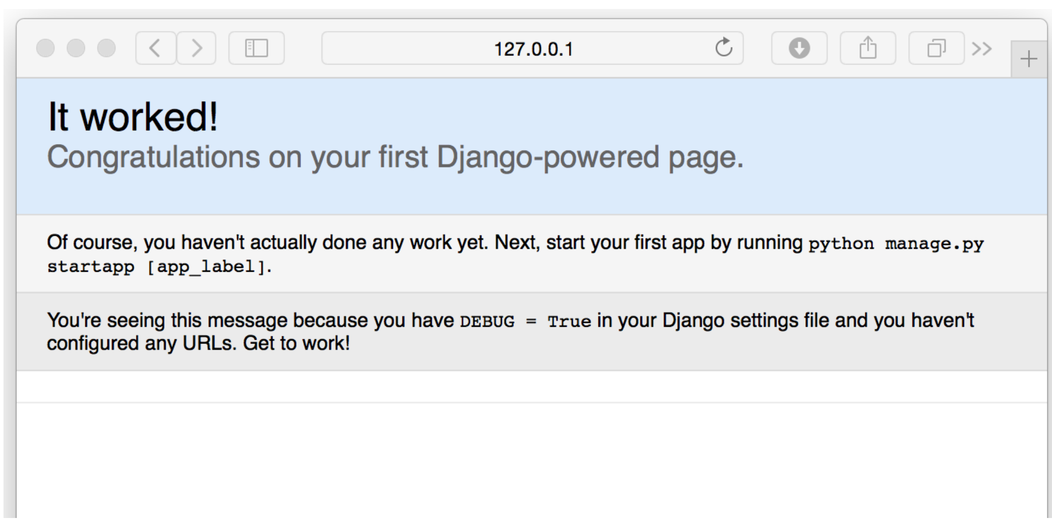


图 3-1：首次运行 Django 开发服务器时看到的页面

开发服务器随时可以停止，只需在终端或命令提示符窗口中按 `CTRL+C` 键。如果想在其他端口上运行开发服务器，或者允许其他设备访问，可以提供可选的参数。例如下述命令：

```
$ python manage.py runserver <your_machines_ip_address>:5555
```

这个命令强制开发服务器在 TCP 端口 `5555` 上响应入站请求。记得把 `<your_machines_ip_address>` 换成你电脑的 IP 地址或 `127.0.0.1`。

★ 不知道自己的 IP 地址？ ★

使用 `0.0.0.0`，Django 能找出你的 IP 地址。不信可以试试：

```
$ python manage.py runserver 0.0.0.0:5555
```

设置端口时，不要使用 80 或 8080，这一般是为 HTTP 保留的。此外，低于 1024 的端口是操作系统专属的。[1]

虽然部署应用时不会使用这个轻量级的开发服务器，但是能让同一网络中的其他设备访问你的应用还是有必要的。使用你的设备的 IP 地址运行服务器能让他人通过 *http://<your_machines_ip_address>:<port>* 访问你的应用。当然，这要看你的网络是怎么配置的，可能要设置代理服务器或防火墙。如果无法远程访问开发服务器，请向网络管理员寻求帮助。

3.3 创建 Django 应用

一个 Django 项目中包含一系列配置和应用，这些在一起共同构成一个完整的 Web 应用或网站。这样做便于运用优秀的软件工程实践。把一个 Web 应用分解为多个小应用的好处是，可以把那些小应用放到别的 Django 项目中，无需做多少改动就能使用。

一个 Django 应用完成一件特殊的任务。一个网站需要多少应用，要视其功能而定。例如，一个项目中可能包含一个投票应用、一个注册应用和一个与内容有关的应用。在另一个项目中，我们可能想复用投票和注册应用，因此可以把它们拿过来用。稍后再详细说明。下面创建 Rango 应用。

在 Django 项目所在的目录（例如 *<workspace>/tango_with_django_project*）中执行下述命令：

```
$ python manage.py startapp rango
```

`startapp` 目录在项目的根目录中创建一个新目录，你可能猜到了，这个目录名为 *rango*，其中包含一些 Python 脚本：

- *__init__.py*：与前面那个的作用完全一样；

1. https://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html

- *admin.py*：注册模型，让 Django 为你创建管理界面；
- *apps.py*：当前应用的配置；
- *models.py*：存放应用的数据模型，即数据的实体及其之间的关系；
- *tests.py*：存放测试应用代码的函数；
- *views.py*：存放处理请求并返回响应的函数；
- *migrations* 目录：存放与模型有关的数据库信息。

views.py 和 *models.py* 是任何应用中都有的两个文件，是 Django 所采用的设计模式（即“模型-视图-模板”模式）的主要部分。如果想深入了解模型、视图和模板之间的关系，请阅读 Django 文档。

在动手创建模型和视图之前，必须告诉 Django 项目这个新应用的存在。为此，要修改项目配置目录中的 *settings.py* 文件。打开那个文件，找到 `INSTALLED_APPS` 列表，把 `rango` 添加到末尾：

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
]
```

再次运行开发服务器，确认 Django 识别了这个新应用。如果能正常启动开发服务器，没有任何错误，说明新应用已经成功识别，可以进入下一步了。

★ startapp 的自动操作 ★

使用 `python manage.py startapp` 命令创建应用时，Django 可能会把新应用的名称自动添加到 *settings.py* 中的 `INSTALLED_APPS` 列表里。尽管如此，在继续之前自己再检查一下也没什么错。

3.4 编写视图

创建好 Rango 应用后，下面编写一个简单的视图。这是我们编写的第一个视图，简单起见，暂不

使用模型或模板，而是把一些文本发回给客户端。

在你选择的 IDE 中打开新建的 *rango* 目录里的 *views.py* 文件。把 `# Create your views here` 这行注释删掉，得到一个空文件。

然后，写入下述代码：

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Rango says hey there partner!")
```

下面分析一下这三行代码，看这个简单的视图是如何运作的：

- 首先，从 `django.http` 模块中导入 `HttpResponse` 对象。
- 在 *views.py* 文件中，一个函数就是一个视图。这里我们只编写了一个视图，即 `index`。
- 视图函数至少有一个参数，即一个 `HttpRequest` 对象，它也在 `django.http` 模块中。按约定，这个参数名为 `request`，不过你可以根据自己的意愿随意使用其他名称。
- 视图必须返回一个 `HttpResponse` 对象。简单的 `HttpResponse` 对象的参数是一个字符串，表示要发给客户端的页面内容。

有了视图还不行，为了让用户能访问视图，要把一个统一资源定位地址（Uniform Resource Locator，URL）映射到视图上。

为此，打开项目配置目录中的 *urls.py* 文件，在 `urlpatterns` 中添加一行代码：

```
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^admin/', admin.site.urls),
]
```

新加的那行代码把根 URL 映射到 `rango` 应用的 `index` 视图上。启动开发服务器（`python manage.py runserver`），访问 *http://127.0.0.1:8000* 或你指定的其他地址。你将看到 `index` 视图渲染的输出。

3.5 映射 URL

为了提升模块化程度，我们可以换种方式把入站 URL 映射到视图上，而不直接在项目层设置。首先，要修改项目的 *urls.py* 文件，把针对 Rango 应用的请求交给 Rango 应用处理。然后，在 Rango 应用中指定如何处理请求。

首先，打开项目配置目录中的 *urls.py* 文件。相对 *workspace* 目录而言，这个文件的地址是 *<workspace>/tango_with_django_project/tango_with_django_project/urls.py*。把 `urlpatterns` 列表改成下面这样：

```
from django.conf.urls import url
from django.contrib import admin
from django.conf.urls import include
from rango import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^rango/', include('rango.urls')),
    # 上面的映射把以 rango/ 开头的 URL 交给 rango 应用处理
    url(r'^admin/', admin.site.urls),
]
```

可以看出，`urlpatterns` 是个 Python 列表。新增的映射寻找能匹配 `^rango/` 模式的 URL。遇到这样的 URL 时，`rango/` 后面的部分传给 Rango，由 `rango.urls` 处理。这一步是通过 `django.conf.urls` 模块中的 `include()` 函数实现的。

这是一种分段处理 URL 字符串的方式，如图 3-2 所示。这里，完整的 URL 先去掉域名，余下的部分（`rango/`）传给 tango_with_django 项目，找到匹配的映射后，再把 `rango/` 去掉，把空字符串传给 `rango` 应用处理。



图 3-2：分段处理 URL，域名后的不同的部分由不同的 *url.py* 文件处理

根据上述设置，我们要在 `rango` 应用的目录中新建 *urls.py* 文件，让它处理余下的 URL（即把空字符串映射到 `index` 视图上）：

```
from django.conf.urls import url
from rango import views
```

```python
urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

这段代码先导入 Django 处理 URL 映射的函数和 Rango 应用的 `views` 模块，然后在 `urlpatterns` 列表中调用 `url` 函数映射 `index` 视图。

本书所指的 URL 字符串，都是去掉主机地址后的部分。主机地址是指向 Web 服务器的地址或域名，例如 *http://127.0.0.1:8000* 或 *http://www.tangowithdjango-china.com*。去掉主机地址后，Django 便只需处理 URL 中余下的部分。例如，对 *http://127.0.0.1:8000/rango/about/* 这个 URL 来说，Django 得到的 URL 字符串是 */rango/about/*。

上述代码中的 URL 映射调用 Django 的 `url()` 函数，其第一个参数是正则表达式 `^$`。这个正则表达式匹配空字符串，因为 `^` 表示开头，`$` 表示结尾，而且二者之间没有任何内容，所以只能匹配空字符串。用户访问的 URL，只要匹配这个模式，Django 就会调用 `views.index()` 视图。你可能觉得匹配空 URL 没有什么意义，那为什么要这样做呢？还记得吗，匹配 URL 模式时，只会考虑原 URL 的一部分。Django 先使用项目的 URL 模式处理 URL 字符串（`rango/`），去掉 `rango/` 部分之后得到空字符。然后把空字符串传给 Rango 应用，交给 *rango/urls.py* 中的 URL 模式处理。

传给 `url()` 函数的下一个参数是 `index` 视图，指明处理入站请求的函数。后面的 `name` 参数是可选的，这里把它设为字符串 `'index'`。为 URL 命名的目的是反向解析 URL，即通过名称引用 URL 映射，而不直接使用 URL。讲到模板时再说明这一点。如果想深入了解这个话题，请阅读 Django 文档。

现在，重启 Django 开发服务器，然后访问 *http://127.0.0.1:8000/rango/*。如果一切正常，你应该能看到文本“Rango says hey there partner!”，如图 3-3 所示。

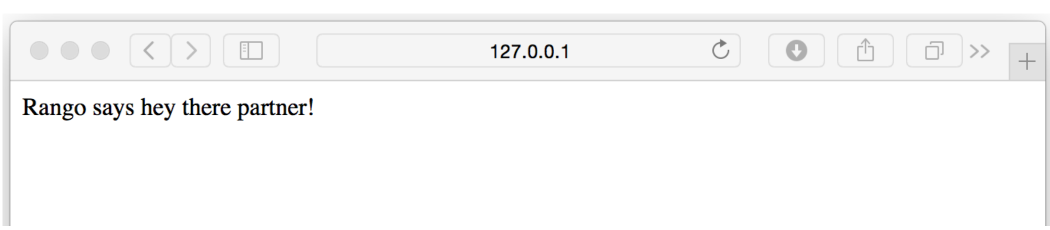


图 3-3：Web 浏览器中显示着首个由 Django 驱动的网页

每个应用中都可以有一些 URL 映射。一开始，URL 映射十分简单，不过随着开发的深入，我们将添加更多复杂的参数化 URL 映射。

你要理解 Django 处理 URL 的方式。现在你可能还有点迷糊，不过用得多了，终究会明白的。如

果想深入了解 URL 映射，想看更多的示例，请阅读 Django 文档。

★ 正则表达式 ★

Django 的 URL 模式使用正则表达式匹配 URL，因此你要熟练使用 Python 的正则表达式。Python 官方文档中有一篇关于正则表达式的指南，此外也可以查看 Regex Cheat Sheet 网站中的速查表。

如果使用版本控制系统，现在是提交改动的好时机。

3.6 基本流程

本章内容可以总结为一系列操作。这一节给出我们所执行的两个任务的操作步骤。如果以后记不得了，可以随时翻阅。

创建 **Django** 项目

1. 执行 `python django-admin.py startproject <name>` 命令，其中 `<name>` 是想创建的项目名称。

创建 **Django** 应用

1. 执行 `python manage.py startapp <appname>` 命令，其中 `<appname>` 是想创建的应用名称。
2. 把应用名称添加到项目配置目录中的 *settings.py* 文件里，放到 `INSTALLED_APPS` 列表的末尾，告诉 Django 项目这个应用的存在。
3. 在项目的 *urls.py* 文件中添加一个映射，指向新建的应用。
4. 在应用的目录中新建 *urls.py* 文件，把入站 URL 与视图对应起来。
5. 在应用的 *view.py* 文件中编写所需的视图，确保视图返回一个 `HttpResponse` 对象。

</> 练习

我们创建了一个 Django 项目，而且把新建的应用运行起来了。请试着完成以下练习，巩固所学的知识。走到这一步不简单，是学习 Django 过程中的一个重要里程碑。编写视图并把 URL 映射到视图上是开发更加复杂的 Web 应用所必须迈出的第一步。

- 回顾本章的内容，确保自己理解了 URL 是如何映射到视图上的。
- 再编写一个视图函数，名为 `about`，返回"Rango says here is the about page."。
- 把这个视图映射到 URL */rango/about/* 上。只需编辑 Rango 应用的 *urls.py* 文件。记住，`/rango/` 部分由项目的 *urls.py* 文件处理。
- 修改 `index` 视图中的 `HttpResponse` 对象，加入一个指向关于页面的链接。
- 在 `about` 视图的 `HttpResponse` 对象中添加一个指向主页的链接。
- 既然开始阅读本书了，那就在 Twitter 上关注 @tangowithdjango 吧，告诉我们你学的怎么样。

★ 提示 ★

如果你做不出上述练习，希望下面的提示能给你一点启发。

- 在 *views.py* 文件中定义一个函数，`def about(request):`，让它返回一个 `HttpResponse` 对象，在参数中指明要返回的 HTML。
- 匹配 URL *about/* 的正则表达式是 `r'^about/'`。在 *rango/urls.py* 文件中为 `about()` 视图增加一个映射。
- 修改 `index()` 视图，添加一个指向 `about` 视图的链接。简单起见，现在可以这样写：`Rango says hey there partner! <br/> <a href='/rango/about/'>About</a>.`。
- 同样，在 `about()` 视图中添加指向首页的链接：`<a href="/rango/">Index</a>`。
- 如果你还没读过 Django 官方教程，现在请阅读第一部分。

4 模板和媒体文件

本章介绍 Django 的模板引擎，并说明如何在应用中伺服静态文件和媒体文件。

4.1 使用模板

目前，我们只把一个 URL 映射到一个视图上。然而，Django 框架采用的是“模型-视图-模板”架构。这一节说明模板与视图的关系，后面几章再讨论如何与模型联系起来。

为什么使用模板？网站中的不同页面通常使用相同的布局，提供通用的页头（header）和页脚（footer），为用户呈现导航，体现一种一致性。Django 模板能让开发者轻易实现这样的设计要求，而且还能把应用逻辑（视图代码）与表现（应用的外观）区分开。本章将创建一个简单的模板，用于生成 HTML 页面，交由 Django 视图调度。第 5 章将更近一步，结合模型动态分发数据。

★ 总结：模板是什么？ ★

Django 的模板可以理解为构建完整的 HTML 页面所需的骨架。模板中有静态内容（不变的部分），也有特殊的句法（动态内容，即模板标签）。Django 视图会把动态内容替换成真正的数据，生成最终的 HTML 响应。

配置模板目录

若想在 Django 应用中使用模板，要创建两个目录，用于存放模板文件。

在 Django 项目配置目录（*<workspace>/tango_with_django_project/*）中创建一个名为 *templates* 的目录。注意，这个目录要与项目的 *manage.py* 脚本放在同一级。在这个新目录中再创建一个目

录，名为 *rango*。我们将在 *<workspace>/tango_with_django_project/templates/rango/* 目录中存放 Rango 应用的模板。

★ 以合理的方式组织模板 ★

建议把各应用的模板放在单独的子目录中。这就是我们在 *templates* 目录中创建 *rango* 子目录的原因。如果想打包应用，把它分发给其他开发者，这样就便于区分模板属于哪个应用。

接下来要告诉 Django 你把模板放在什么位置。打开项目的 *settings.py* 文件，找到 `TEMPLATES`。如果项目是使用 Django 1.9 创建的，`TEMPLATES` 的默认内容如下：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

为了告诉 Django，模板在何处，我们要修改 `DIRS` 列表（默认为空）。把这个键值对改成下面这样：

```
'DIRS': ['<workspace>/tango_with_django_project/templates']
```

注意，这里要使用*绝对路径*（absolute path）。然而，与团队成员协作，或者换台电脑的话，这就是个问题了。用户名和磁盘结构变了，`<workspace>` 目录的路径也就不一样了。这个问题的一种解决方法是列出每个可能的路径，例如：

```
'DIRS': [ '/Users/leifos/templates',
          '/Users/maxwelld90/templates',
          '/Users/davidm/templates', ]
```

可是这样做也有诸多问题。首先，每增加一个团队成员就要增加一个路径。其次，如果换成其他操作系统（例如 Windows），斜线可能都要换种形式。

◆ 不要硬编码路径 ◆

硬编码路径是自找麻烦。硬编码路径是软件工程的一种反模式，会导致项目不易移植，也就是说在一台电脑中运行好好的，换台设备可能就会出错。

动态路径

更好的方法是使用 Python 内置的函数自动找出 *templates* 目录的路径。这样无论你把 Django 项目的代码放在何处，最终都能得到一个绝对路径。因此，项目的可移植性更高。

settings.py 文件的顶部有个名为 `BASE_DIR` 的变量，它的值是 *settings.py* 文件所在目录的路径。这里用到了 Python 的特殊属性 `__file__`，它的值是所在文件的绝对路径。调用 `os.path.dirname()` 的作用是获取 *settings.py* 文件所在目录的绝对路径，再调用一次 `os.path.dirname()` 又去掉一层，因此 `BASE_DIR` 最终的值是 *<workspace>/tango_with_django_project/*。如果你还不太理解这个过程，可以把下面几行代码放到 *settings.py* 文件中：

```
print(__file__)
print(os.path.dirname(__file__))
print(os.path.dirname(os.path.dirname(__file__)))
```

有了 `BASE_DIR` 之后，我们便可以轻易引用 Django 项目中的文件和目录。我们可以定义一个名为 `TEMPLATE_DIR` 的变量，指向 `templates` 目录的位置。这里还要使用 `os.path.join()` 函数拼接多个路径片段。`TEMPLATE_DIR` 变量的定义如下：

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

我们使用 `os.path.join()` 函数把 `BASE_DIR` 变量和 `'templates'` 字符串拼接起来，得到 `<workspace>/tango_with_django_project/templates/`。如此一来，我们便可以使用 `TEMPLATE_DIR` 变量替代前面在 `TEMPLATES` 中硬编码的路径。把 `DIRS` 键值对改成下面这样：

```
'DIRS': [TEMPLATE_DIR, ]
```

★ 为什么命名为 TEMPLATE_DIR？ ★

我们在 *settings.py* 文件的顶部定义了一个名为 TEMPLATE_DIR 的变量，采用这个名称是因为它能表明变量的作用，而且基本不会修改。在较复杂的 Django 项目中，可以在 DIRS 列表中指定多个模板目录，但是本书用这一个就够了。

◆ 拼接路径 ◆

拼接系统路径时一定要使用 os.path.join() 函数，这样为的是使用正确的路径分隔符。Unix 操作系统（及其衍生系统）使用正斜线（/）分隔目录，而 Windows 操作系统使用反斜线（\）。如果直接使用斜线，更换操作系统后可能导致路径错误，从而降低项目的可移植性。

添加一个模板

模板目录和路径设置好之后，在 *templates/rango/* 目录中创建一个文件，命名为 *index.html*。在这个新文件中写入下述 HTML 代码。

```
<!DOCTYPE html>
<html>

    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>
        <div>
            hey there partner! <br />
            <strong>{{ boldmessage }}</strong><br />
        </div>
        <div>
            <a href="/rango/about/">About</a><br />
        </div>
    </body>

</html>
```

这段 HTML 代码很好理解，最终得到的 HTML 页面将向用户打个招呼。你可能注意到了，这里有些内容不是 HTML，而是 `{{ boldmessage }}` 形式。这是 Django 模板变量。我们可以为这样的变量设值，这样渲染模板后便会显示我们设定的值。稍后再做。

为了使用这个模板，我们要调整一下前面编写的 `index()` 视图，不再分发一个简单的响应对象，而是分发这个模板。

打开 *rango/views.py* 文件，看看文件顶部有没有下面这个 `import` 语句。如果没有，加上。

```
from django.shortcuts import render
```

然后根据下述代码片段修改 `index()` 视图函数。各行代码的作用参见注释。

```
def index(request):
    # 构建一个字典，作为上下文传给模板引擎
    # 注意，boldmessage 键对应于模板中的 {{ boldmessage }}
    context_dict = {'boldmessage': "Crunchy, creamy, cookie, candy, cupcake!"}

    # 返回一个渲染后的响应发给客户端
    # 为了方便，我们使用的是 render 函数的简短形式
    # 注意，第二个参数是我们想使用的模板
    return render(request, 'rango/index.html', context=context_dict)
```

首先，构建一个字典，设定要传给模板的数据。然后，调用 `render()` 辅助函数。这个函数的参数是 `request` 对象、模板的文件名和上下文字典。`render()` 函数将把上下文字典中的数据代入模板，生成一个完整的 HTML 页面，作为 `HttpResponse` 对象返回，分发给 Web 浏览器。

★ 模板上下文是什么？ ★

Django 的模板系统加载模板文件时会创建一个模板上下文。简单而言，模板上下文是一个 Python 字典，把模板变量名映射到值上。在前面的模板中，有个名为 `boldmessage` 的模板变量。在修改后的 `index()` 视图中，我们把 `Crunchy, creamy, cookie, candy, cupcake!` 字符串赋给模板变量 `boldmessage`。这样，渲染模板时，模板中的 `{{ boldmessage }}` 将会替换为字符串 `Crunchy, creamy, cookie, candy, cupcake!`。

我们已经更新视图，用上了模板。现在启动 Django 开发服务器，然后访问 *http://127.0.0.1:8000/rango/*。你应该能看到这个简单的 HTML 模板渲染出来了，如图 4-1 所示。

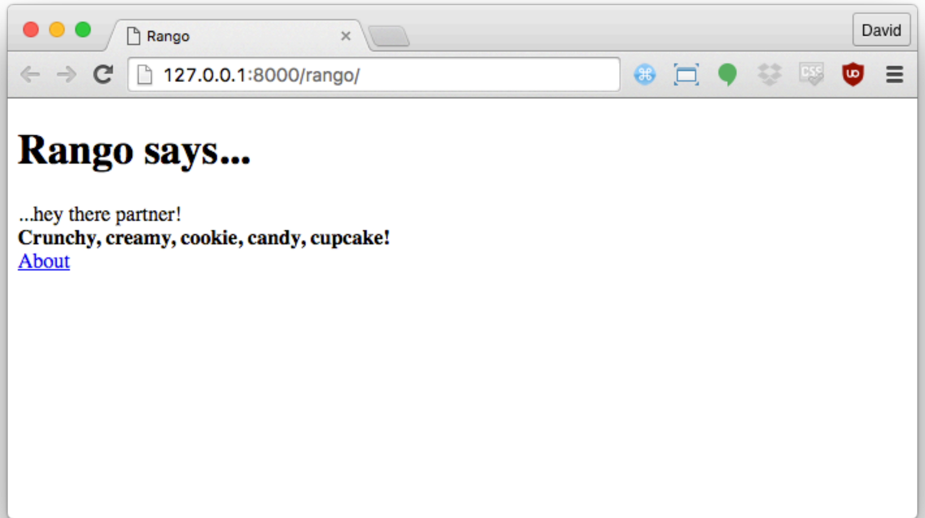


图 4-1：一切正常时应该看到的页面。注意加粗的文本，即“Crunchy, creamy, cookie, candy, cupcake!”，它们取自视图，通过模板渲染出来。

如果没看到上述页面，读一下错误消息，看看是什么地方出错了，再次确认前面所做的改动。常见的问题之一是，*settings.py* 文件中的模板路径设置的不对。可以在 *settings.py* 文件中使用 `print` 语句输出 `BASE_DIR` 和 `TEMPLATE_DIR` 的值，确认一切是否正常。

这个示例演示了如何在视图中使用模板。然而，我们只用了 Django 模板引擎丰富功能中的一点皮毛。本书后文还将使用更复杂的模板功能。如果想深入了解模板，请阅读 Django 文档。

4.2 伺服静态文件

尽管我们用上了模板，但是不得不承认，Rango 应用现在还有点简陋，没有样式也没有图像装饰。为了改善这种状况，我们可以在 HTML 模板中引用其他文件，例如*层叠样式表*（Cascading Style Sheet，CSS）、JavaScript 和图像。这些是*静态文件*（static file），因为它们不是由 Web 服务器动态生成的，而是原封不动发给 Web 浏览器。本节说明 Django 伺服静态文件的方式，以及如何在模板中添加一个图像。

配置静态文件目录

首先要指定一个目录，用于存放静态文件。在项目配置目录中新建一个目录，名为 *static*，然后在

static 目录中再新建一个目录，名为 *images*。确保 *static* 目录与前面创建的 *templates* 目录位于同一级。

然后，在 *images* 目录中放一个图像。如下图所示，我们选择的是动画电影《兰戈》中变色龙兰戈的图片。

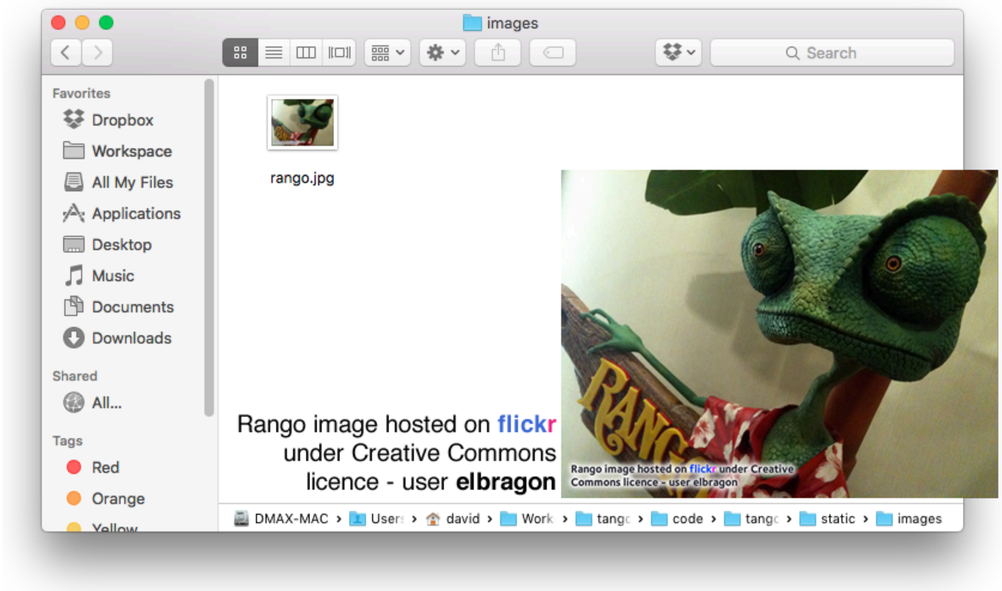


图 4-2：把变色龙兰戈的图片放到 *static/images* 目录中

与前面的 *templates* 目录一样，我们要告诉 Django 这个 *static* 目录的路径。为此，还要编辑项目的 *settings.py* 模块。在这个文件中，我们要定义一个变量，指向 *static* 目录，并在一个数据结构中告诉 Django 这个目录的路径。

首先，在 *settings.py* 文件的顶部定义一个变量，名为 `STATIC_DIR`。为了把所有路径放在一起，可以把 `STATIC_DIR` 放在 `BASE_DIR` 和 `TEMPLATES_DIR` 下面。`STATIC_DIR` 的值也应该使用前面用过的 `os.path.join`，不过这一次是指向 *static* 目录，如下所示：

```
STATIC_DIR = os.path.join(BASE_DIR, 'static')
```

这样得到的是一个绝对路径，指向 *<workspace>/tango_with_django_project/static/*。定义好这个变量之后，还要创建一个数据结构，名为 `STATICFILES_DIRS`。这个数据结构的值是一系列路径，让 Django 在其中寻找要伺服的静态文件。默认情况下，*settings.py* 文件中没有这个列表。在创建之前，确认这个列表确实不存在。如果定义两次，Django 会混淆的，你自己也是一样。

本书只在一个位置存放项目的静态文件，即 `STATIC_DIR` 定义的路径。因此，我们可以像下面这样

设置 STATICFILES_DIRS：

```
STATICFILES_DIRS = [STATIC_DIR, ]
```

★ 保持 *settings.py* 整洁有序 ★

最好保持 *settings.py* 模块整洁有序。不要随意乱放，要有组织。把定义目录位置的变量放在模块的顶部，这样便于查找。把 STATICFILES_DIRS 放在与静态文件相关的那部分（靠近底部）。这样对你和其他协作者来说都方便。

最后，检查 *settings.py* 模块中有没有定义 STATIC_URL 变量。如果没有，像下面那样定义。注意，Django 1.9 默认把这个变量放在靠近底部的位置，因此你可能要向下拉滚动条才能找到。

```
STATIC_URL = '/static/'
```

我们做了这么多，还没说为什么要这么做。简单来说，STATIC_DIR 和 STATICFILES_DIRS 两个变量设定静态文件在电脑中的位置；STATIC_URL 变量则指定启动 Django 开发服务器后通过什么 URL 访问静态文件。例如，把 STATIC_URL 设为 /static/ 后，我们可以通过 *http://127.0.0.1:8000/static/* 访问里面的静态内容。前两个变量相当于服务器端的位置，而第三个变量是客户端访问静态内容的位置。

</> 测试配置

现在，请测试一下一切是否配置正确。启动 Django 开发服务器，在浏览器中访问 *rango.jpg* 图像试试。如果把 STATIC_URL 设为 /static/，而 *rango.jpg* 的位置是 *images/rango.jpg*，想一想应该在 Web 浏览器中输入什么 URL？

在继续阅读之前，请仔细想一想。想不出来也没关系，后文会告诉你。

◆ 别忘了斜线 ◆

设置 STATIC_URL 时，确保 URL 的末尾有一个正斜线（例如，是 /static/，而不是 /static）。根据 Django 文档，缺少末尾的斜线会导致一系列问题。在末尾加上斜线的目的是把 URL 的根部（例如 /static/）与要伺服的静态内容（例如 images/rango.jpg）区分

开。

★ 伺服静态内容 ★

在开发环境中使用 Django 开发服务器伺服静态文件没问题，但是在开发环境中却不合适。Django 文档中有关于在生产环境中部署静态文件的更多信息。第 18 章将深入讨论这个问题。

如果你没想出应该通过哪个 URL 访问那个图像，请在 Web 浏览器中输入 *http://127.0.0.1:8000/static/images/rango.jpg*。

在模板中引用静态文件

我们已经做好设置，Django 项目能处理静态文件了。现在可以在模板中利用静态文件改进外观及增添功能了。

下面说明如何引用静态文件。打开 *<workspace>/templates/rango/* 目录中的 *index.html* 模板，参照下述代码修改。为了方便查找，新增的行旁边有注释。

```
<!DOCTYPE html>

{% load staticfiles %} <!-- 新增 -->

<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>

        <div>
            hey there partner! <br />
            <strong>{{ boldmessage }}</strong><br />
        </div>
```

```
        <div>
            <a href="/rango/about/">About</a><br />
            <img src="{% static "images/rango.jpg" %}"
                 alt="Picture of Rango" /> <!-- 新增 -->
        </div>
    </body>

</html>
```

新增的第一行（`{% load staticfiles %}`）告诉 Django 模板引擎，我们将在模板中使用静态文件。这样便可以在模板中使用 `static` 模板标签引入静态目录中的文件。`static "images/rango.jpg"` 告诉 Django，我们想显示静态目录中名为 *images/rango.jpg* 的图像。`static` 标签会在 *images/rango.jpg* 前加上 `STATIC_URL` 指定的 URL，得到 */static/images/rango.jpg*。Django 模板引擎生成的 HTML 如下：

```
<img src="/static/images/rango.jpg" alt="Picture of Rango" />
```

建议为图像设定替代文本，以防图片由于某些原因无法加载。替代文本通过 `img` 标签的 `alt` 属性指定，效果如下。

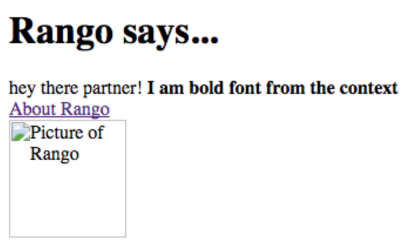


图 4-3：找不到图像时显示 `alt` 属性的值

修改视图之后，启动 Django 开发服务器，然后访问 *http://127.0.0.1:8000/rango*。如果一切正常，应该会看到类似图 4-4 中的网页。

★ 模板中的 `<!DOCTYPE>` ★

HTML 模板的第一行始终是 `DOCTYPE` 声明。如果把 `{% load staticfiles %}` 放在前面，渲染得到的 HTML 中，`DOCTYPE` 声明前面将出现空白。多出的空白会导致 HTML 标记无法通过验证。

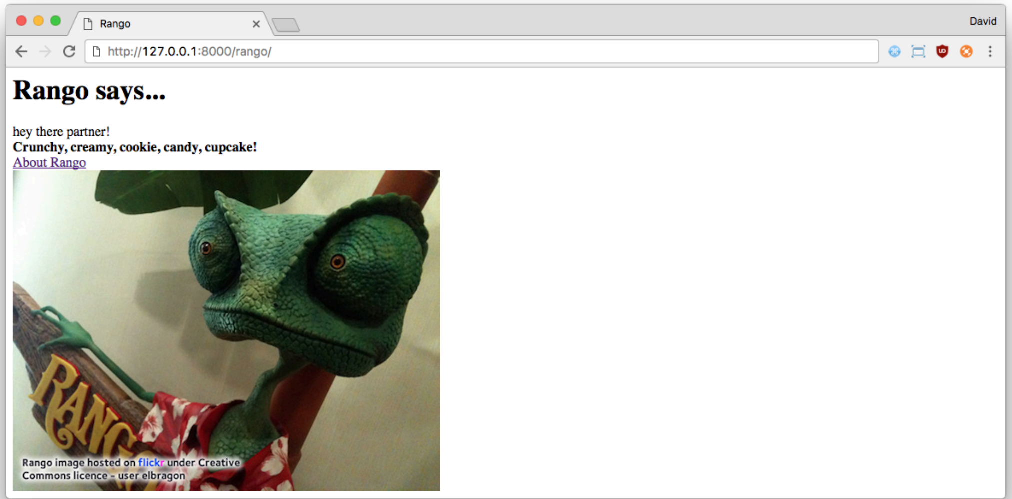


图 4-4：我们的第一个模板中显示着变色龙兰戈的图片

★ 加载其他静态文件 ★

只要想在模板中引用静态文件，就可以使用 {% static %} 模板标签。下述代码片段说明如何在模板中引入 JavaScript、CSS 和图像。

```
<!DOCTYPE html>
{% load staticfiles %}

<html>

    <head>
        <title>Rango</title>
        <!-- CSS -->
        <link rel="stylesheet" href="{% static "css/base.css" %}" />
        <!-- JavaScript -->
        <script src="{% static "js/jquery.js" %}"></script>
    </head>

    <body>
        <!-- 图像 -->
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </body>
</html>
```

显然，*static* 目录中要有你引用的静态文件。如果引用的文件不存在，或者未正确引用，Django 开发服务器在控制台中的输出将报告 HTTP 404 错误。你可以引用一个不存在的文件试试。注意下述输出中的最后一条，HTTP 状态码是 404。

```
[10/Apr/2016 15:12:48] "GET /rango/ HTTP/1.1" 200 374
[10/Apr/2016 15:12:48] "GET /static/images/rango.jpg HTTP/1.1" 304 0
[10/Apr/2016 15:12:52] "GET /static/images/not-here.jpg HTTP/1.1" 404 0
```

如果想深入了解引入静态文件的方式，请阅读 Django 文档。

4.3 伺服媒体文件

应用中的静态文件可以理解为不变的文件。不过，有时还要使用可变的媒体文件（media file）。这类文件可由用户或管理员上传，因此可能会变化。比如说，用户的头像就是媒体文件，电商网站中的商品图片也是媒体文件。

为了能伺服媒体文件，我们要修改 Django 项目的设置。这一节说明具体需要做哪些设置，但暂不测试，等到实现用户上传头像功能时再做检查。

★ 伺服媒体文件 ★

与静态文件一样，在 Django 开发环境中也能检查设置是否正确。当然，开发环境中伺服媒体文件的方法极其不适合在生产环境中使用，因此部署后应该寻找其他方式托管应用的媒体文件。详情参见第 18 章。

修改 *settings.py*

首先，打开 Django 项目配置目录中的 *settings.py* 模块。我们将在这个文件中添加一些内容。与静态文件一样，媒体文件也放在文件系统中专门的一个目录中。因此，要告诉 Django 这个目录的位置。

在 *settings.py* 模块的顶部，找到 `BASE_DIR`、`TEMPLATE_DIR` 和 `STATIC_DIR` 变量。在它们后面加上 `MEDIA_DIR`：

```
MEDIA_DIR = os.path.join(BASE_DIR, 'media')
```

这一行告诉 Django，媒体文件将上传到 Django 项目根目录中的 *media* 目录里，即 *<workspace>/tango_with_django_project/media/*。正如前文所说，把路径相关的变量放在 *settings.py* 模块的顶部便于以后修改。

然后在 *settings.py* 中找个地方添加两个变量：`MEDIA_ROOT` 和 `MEDIA_URL`。Django 伺服媒体文件时会用到这两个变量。

```
MEDIA_ROOT = MEDIA_DIR
MEDIA_URL = '/media/'
```

◆ 同样，别忘了斜线 ◆

与 `STATIC_URL` 变量一样，`MEDIA_URL` 变量的末尾要有一条正斜线（例如，是 `/media/`，而不是 `/media`）。在末尾加上斜线的目的是把 URL 的根部（例如 `/media/`）与用户上传的内容区分开。

`MEDIA_ROOT` 变量告诉 Django 在哪里寻找上传的媒体文件，`MEDIA_URL` 变量则指明通过什么 URL 伺服媒体文件。这样设置之后，上传的 *cat.jpg* 文件在 Django 开发服务器中将通过 *http://localhost:8000/media/cat.jpg* 访问。

后面在模板中引入上传的内容时，如果能方便引用 `MEDIA_URL` 路径就好了。为了方便这样的操作，Django 提供了模板上下文处理器。严格来说，现在无需这么做，但是趁机加上也没关系。

找到 *settings.py* 文件中的 `TEMPLATES` 列表，里面嵌套着 `context_processors` 列表。在 `context_processors` 列表中添加一个处理器，`django.template.context_processors.media`。添加之后，`context_processors` 列表应该类似下面这样：

```
'context_processors': [
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
    'django.template.context_processors.media'
],
```

调整 URL

在开发环境中伺服媒体文件的最后一步是，让 Django 使用 `MEDIA_URL` 伺服媒体内容。打开项目的 *urls.py* 模块，修改 `urlpatterns` 列表，调用 `static()` 函数：

```
urlpatterns = [
    ...
    ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

此外，还要在 *urls.py* 模块的顶部添加下述 `import` 语句：

```
from django.conf import settings
from django.conf.urls.static import static
```

现在可以通过 */media/* URL 访问 *media* 目录中的媒体文件了。

4.4 基本流程

至此，你应该知道如何设置和创建模板，知道如何在视图中使用模板，知道如何设置并让 Django 开发服务器伺服静态媒体文件，以及如何在模板中引入图像了。本章的知识可不少！

本章的重点是知道如何创建模板，并在 Django 视图中使用模板。这其中涉及好几步，不过多做几次就会习惯的。

1. 首先，创建要使用的模板，保存到 *templates* 目录中（在项目的 *settings.py* 模块中设定）。模板中可以使用 Django 模板变量（例如 `{{ variable_name }}`）或模板标签。模板变量的值在相应的视图中设定。
2. 在应用的 *views.py* 文件中找到所需的视图，或者新建一个。
3. 把视图相关的逻辑写在视图函数中。例如，从数据库中检索数据，存到列表中。
4. 在视图中构建一个字典，通过模板上下文传给模板引擎。
5. 使用 `render()` 辅助函数生成响应。这个函数的参数是请求对象、模板文件名和上下文字典。
6. 如果还没把视图映射到 URL 上，修改项目的 *urls.py* 文件和应用的 *urls.py* 文件。

在网页中引入静态文件的步骤也很重要，应该熟练掌握。具体方法参见下述步骤。

1. 把想用的静态文件放到项目的 *static* 目录中。这个目录在项目的 *settings.py* 模块中的 `STATICFILES_DIRS` 列表中设定。
2. 在模板中引用静态文件。例如，图像通过 `<img />` 标签插入 HTML 页面。
3. 记得在模板中加上 `{% load staticfiles %}`，然后使用 `{% static "<filename>" %}` 标签引用静态文件。把 `<filename>` 替换成图像或其他资源的路径。只要想引用静态文件，使用 `static` 模板标签。

伺服媒体文件的步骤与伺服静态文件差不多。

1. 把媒体文件放到项目的 *media* 目录中。这个目录由项目的 *settings.py* 模块中的 `MEDIA_ROOT` 变量设定。
2. 在模板中使用 `{{ MEDIA_URL }}` 上下文变量引用媒体文件。例如，引用上传的图像 *cat.jpg*：`<img src="{{ MEDIA_URL}}cat.jpg" />`。

</> 练习

请完成以下练习，巩固本章所学的知识。

- 让关于页面也使用模板渲染，模板名为 *about.html*。
- 在 *about.html* 模板中引入一个图片（存储在项目的 *static* 目录中）。
- 在关于页面中添加一行：`This tutorial has been put together by <your-name>.`。
- 在 Django 项目配置目录中新建一个目录，命名为 *media*。从网上下载一张猫的图片，保存到 *media* 目录中，命名为 *cat.jpg*。
- 在关于页面中添加一个 `<img>` 标签，显示那个猫的图片，确保媒体文件能正确伺服。加上首页中的兰戈图片，现在你的应用既可以伺服静态文件，也可以伺服媒体文件。

★ 静态文件 vs. 媒体文件 ★

从名称可以看出，静态文件是不变的，是网站的核心构成组件。而媒体文件是用户提供的，时常变化。

样式表是静态文件，定义网页的外观。而用户的头像是媒体文件，用户在注册账户时上传。

5 模型与数据库

说到数据库，你往往会想到结构化查询语言（Structured Query Language，SQL），这是从数据库中查询所需数据的常用工具。然而在 Django 中，查询底层数据库这项操作由对象关系映射器（Object Relational Mapper，ORM）负责。在 ORM 中，存储在一个数据库表中的数据封装为一个模型。模型则是描述数据库表中数据的 Python 对象。Django 不直接通过 SQL 查询数据库，而是使用相应的 Python 模型对象操纵。

本章介绍使用 Django 及其 ORM 管理数据的基础知识。你会发现，通过 ORM 添加、修改和删除底层数据库中的数据特别简单，而且把数据库中的数据检索出来提供给 Web 浏览器使用也十分方便。

5.1 Rango 的要求

学习新知识之前，先回顾一下 Rango 应用对数据的要求。详细要求参见前文，这里只简要列出客户的要求。

- Rango 其实是个网页目录，全是指向其他网站的链接。
- 网页有分类，一个分类中有一系列链接。第 1 章说过，我们假定这是一对多关系，如下面的实体-关系图所示。
- 分类有名称，要记录查看次数和点赞次数。
- 网页有标题和 URL，要记录访问次数。



图 5-1：两个主要实体之间的 ER 图

5.2 设置数据库

创建模型之前要设置数据库。新建项目时，Django 已经自动在 *settings.py* 模块中添加了 `DATABASES` 变量，其值类似下面这样：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

对 Rango 应用来说，使用默认值基本就行了。可以看到，`default` 数据库由一个轻量级数据库引擎驱动，即 SQLite（见 `ENGINE` 键）。`NAME` 键的值是数据库文件的路径，默认为 Django 项目根目录中的 `db.sqlite3`。

■ Git 提示 ■

如果你使用 Git，很可能想把数据库文件纳入版本控制。这可不是个好主意，因为如果你与别人协作的话，它们说不定会修改数据库，这会导致无穷尽的冲突。

正确的做法是把 *db.sqlite3* 添加到 *.gitignore* 文件中，在提交和推送时排除数据库文件。**.pyc* 和设备专用的文件也应该忽略。

★ 使用其他数据库引擎 ★

Django 数据库框架支持多种不同的数据库引擎，例如 PostgresSQL、MySQL 和微软的 SQL Server。使用这些数据库引擎时，可以使用 `USER`、`PASSWORD`、`HOST` 和 `PORT` 等键配置。

本书不说明如何使用其他数据库引擎，如果想了解这方面的知识，请到网上查找资源。可以从 Django 文档开始。

SQLite 足够说明 Django ORM 的功能了。如果你的应用大受欢迎、流量激增，或许应该考虑换用更可靠的数据库后端。

5.3 创建模型

在 *settings.py* 模块中配置好数据库之后，下面创建 Rango 应用所需的两个数据模型。Django 应用的模型保存在应用目录中的 *models.py* 模块里。因此，Rango 应用的模型保存在 *rango/models.py* 文件中。

我们要定义两个类，一个类对应一个模型。这两个类都要继承基类 `django.db.models.Model`，分别表示分类和网页。参照下述代码片段定义 `Category` 和 `Page` 模型。

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    def __str__(self):  # Python 2 还要定义 __unicode__
        return self.name


class Page(models.Model):
    category = models.ForeignKey(Category)
    title = models.CharField(max_length=128)
    url = models.URLField()
    views = models.IntegerField(default=0)

    def __str__(self):  # Python 2 还要定义 __unicode__
        return self.title
```

■ 检查 `import` 语句 ■

models.py 模块的顶部应该有 `from django.db import models`。如果没有，自己动手加上。

★ __str__() vs. __unicode__() ★

在 Python 中，`__str__()` 和 `__unicode__()` 方法生成类的字符串表示形式（类似于 Java 中的 `toString()` 方法）。在 Python 2.x 中，`__str__()` 方法以 ASCII 格式表示字符串；如果需要支持 Unicode，还要实现 `__unicode__()` 方法。

在 Python 3.x 中，字符串默认就支持 Unicode，因此只需实现 `__str__()` 方法。

定义模型时，要指定字段及其类型，并提供必须或可选的参数。默认情况下，每个模型都有一个自增整数字段，名为 `id`。这个字段自动分配，用作主键。

Django 内置的字段类型无所不包。下面介绍其中最常用的几个：

- `CharField`：存储字符数据（如字符串）的字段。`max_length` 参数指定最多可存储的字符数。
- `URLField`：类似于 `CharField`，不过是专用于存储 URL 的。也可指定 `max_length` 参数。
- `IntegerField`：存储整数。
- `DateField`：存储 Python `datetime.date` 对象。

★ 其他字段类型 ★

Django 提供的全部字段类型，以及各自必须和可选的参数说明，参见 Django 文档。

每个字段都可指定 `unique` 参数，设为 `True` 时，字段的值在模型对应的底层数据库表中必须是唯一的。看一下前面定义的 `Category` 模型。`name` 字段设为唯一的了，这表明每个分类的名称必须是唯一的。这也意味着，可以把 `name` 字段用作主键。

此外，各字段还可指定其他参数，例如通过 `default='value'` 设定默认值，指定字段的值可以为空（`null=True`）或者不可以为空（`null=False`）。

为了建立模型之间的关系，Django 提供了三个字段类型，分别是：

- `ForeignKey`：用于建立一对多关系。
- `OneToOneField`：用于建立一对一关系。
- `ManyToManyField`：用于建立多对多关系。

在我们定义的模型中，Page 模型中的 category 字段是 ForeignKey 类型。我们在字段的构造方法中指定，与 Category 模型建立一对多关系。

最后，建议实现 __str__() 和（或）__unicode__() 方法。如果不实现，使用 print 打印对象时将显示 <Category: Category object>，这对调试没什么帮助；如果实现的话，打印“Python”分类时将显示 <Category: Python>。这两个方法在管理界面中也用得到，因为 Django 会在管理后台中显示对象的字符串表示形式。

5.4 创建和迁移数据库

在 *models.py* 中定义好模型之后，可以让 Django 施展魔法，在底层数据库中创建表了。为此，Django 提供了迁移工具，让它帮助我们设置和更新数据库，体现模型的改动。例如，添加新字段后可以使用迁移工具更新数据库。

设置

首先，数据库必须预置，即创建数据库及相关的表。请打开终端或命令提示符，进入项目的根目录（*manage.py* 文件所在的目录），执行下述命令。注意，你看到的输出可能与这里不一样。

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

执行这个命令后，项目中安装的所有应用都会更新各自的数据库表。这个命令执行完毕后，Django 项目的根目录中会出现 *db.sqlite3* 文件。

然后，创建一个超级用户（superuser），用于管理数据库。执行下述命令：

```
$ python manage.py createsuperuser
```

这个超级用户在本章后面将用于访问 Django 管理界面。请根据提示输入用户名、电子邮件地址和密码。别忘了记下超级用户的用户名和密码。

创建和更新模型/表

每次修改应用的模型都要通过 *manage.py* 中的 `makemigrations` 命令登记改动。在 Django 项目的根目录中执行下述命令，指明目标为 `rango` 应用：

```
$ python manage.py makemigrations rango

Migrations for 'rango':
  0001_initial.py:
    - Create model Category
    - Create model Page
```

上述命令执行完毕后，*rango/migrations* 目录中会出现一个 Python 脚本，名为 *0001_initial.py*。这个脚本中包含此次迁移创建数据库模式所需的一切信息。

★ 检查底层 SQL ★

如果想查看 Django ORM 为某个迁移执行的底层 SQL，可以执行下述命令：

```
$ python manage.py sqlmigrate rango 0001
```

这里，`rango` 是应用的名称，`0001` 是想查看 SQL 代码的迁移。这样做能更好地理解数据库层具体做了哪些操作，例如创建了什么表。你可能会发现其中包含复杂的数据库模式，比如建立多对多关系时会创建额外的表。

创建好迁移后，要提交到数据库。为此要再次执行 `migrate` 命令。

```
$ python manage.py migrate
```

```
Operations to perform:
  Apply all migrations: admin, rango, contenttypes, auth, sessions
Running migrations:
  Rendering model states... DONE
  Applying rango.0001_initial... OK
```

输出表明在数据库中创建了所需的表。

不过，你可能注意到了，根据客户对 Rango 提出的要求，`Category` 模型还缺少一些字段。别担心，后面会再次通过迁移加上。

5.5 Django 模型和 shell

暂不介绍 Django 管理界面，先来看看 Django shell。这个工具对调试十分有用，可以直接与 Django 模型交互。下面说明如何在 shell 中创建 `Category` 实例。

为了打开 shell，我们要再次在项目的根目录中运行 *manage.py* 脚本。执行下述命令：

```
$ python manage.py shell
```

这个命令启动 Python 解释器，并加载项目的设置。在这个 shell 中可以与模型交互，具体方法参见下述终端会话中的演示。留意笔者加上的注释，了解各命令的作用。注意，Django 1.9 和 1.10 返回的结果稍有不同，下述示例都给出了，并在注释中做了说明。

```
# 从 Rango 应用中导入 Category 模型
>>> from rango.models import Category

# 显示目前的所有分类
>>> print(Category.objects.all())
# 下述输出分别针对 Django 1.9 和 Django 1.10
# 意思是一样的，都表示还未定义分类
[]  # Django 1.9 的输出：一个空列表
<QuerySet []>  # Django 1.10 的输出：一个空 QuerySet 对象

# 创建一个分类对象，存入数据库
>>> c = Category(name="Test")
>>> c.save()

# 再次列出所有分类对象
>>> print(Category.objects.all())
```

```
# 下述输出分别针对 Django 1.9 和 Django 1.10
# 两个输出中都有测试分类
[<Category: Test>]  # Django 1.9
<QuerySet [<Category: Test>]  # Django 1.10

# 退出 Django shell
>>> quit()
```

这里，我们先导入要操纵的模型。然后打印现有的全部分类。因为底层的 `category` 表是空的，所以返回一个空列表。之后，我们创建并保存了一个 `Category` 对象，然后再次打印所有分类。这一次打印发现有个新分类。注意，第二次打印显示了分类的名称，这是 `__str__()` 或 `__unicode__()` 方法的功劳。

</> 阅读官方教程

以上只是简单演示，除此之外，在 Django shell 中可以执行的操作还有很多。如果你还没阅读 Django 官方教程的第二部分，现在是个好时机，请留意在 shell 中与模型交互的更多方法。顺便看一下 Django 文档中列出的模型操作命令。

5.6 配置管理界面

Django 广受欢迎的一个功能是内置的 Web 管理界面，在这里你可以浏览、标记和删除模型实例表示的数据。本节将做些设置，让你在管理界面中查看 Rango 应用的两个模型。

相关的设置很简单。在项目的 *settings.py* 模块中你可能注意到了，有个预装的应用是 `django.contrib.admin`（在 `INSTALLED_APPS` 列表中）。此外，在项目的 *urls.py* 模块中有个匹配 *admin/* 的 URL 模式（在 `urlpattern` 中）。

其实，使用默认的设置基本就行了。执行下述命令，启动 Django 开发服务器：

```
$ python manage.py runserver
```

然后打开 Web 浏览器，访问 *http://127.0.0.1:8000/admin/*。你会看到登录界面。使用创建超级用户时提供的凭据登录。登录后会看到类似图 5-2 所示的界面。

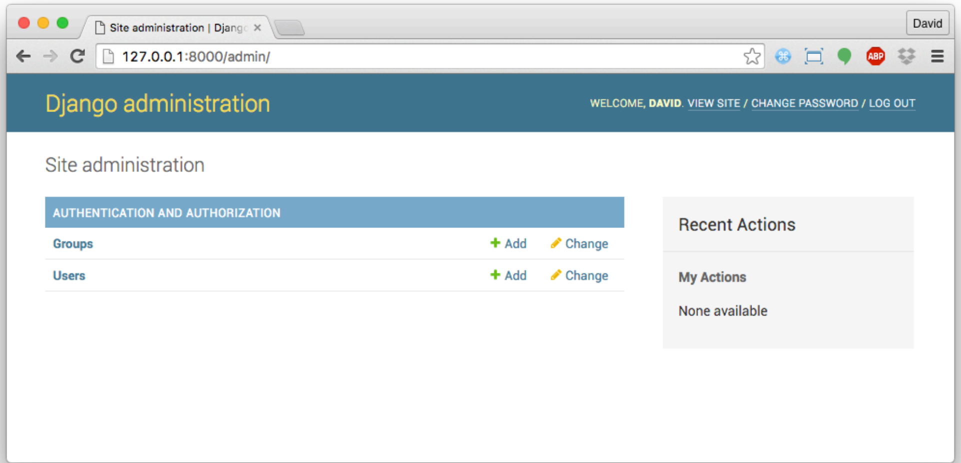


图 5-2：Django 管理界面，没有 Rango 应用的模型

看起来不错，但是没有 Rango 应用的 `Category` 和 `Page` 模型。为了显示这两个模型，我们要给 Django 一些提示。

打开 *rango/admin.py* 文件，注册想在管理界面显示的类。下述代码注册 `Category` 和 `Page` 两个类。

```
from django.contrib import admin
from rango.models import Category, Page

admin.site.register(Category)
admin.site.register(Page)
```

如果以后添加更多模型，只需再调用 `admin.site.register()` 方法。

改好之后，重启 Django 开发服务器，然后再次访问 *http://127.0.0.1:8000/admin/*。现在能看到 `Category` 和 `Page` 模型了，如图 5-3 所示。

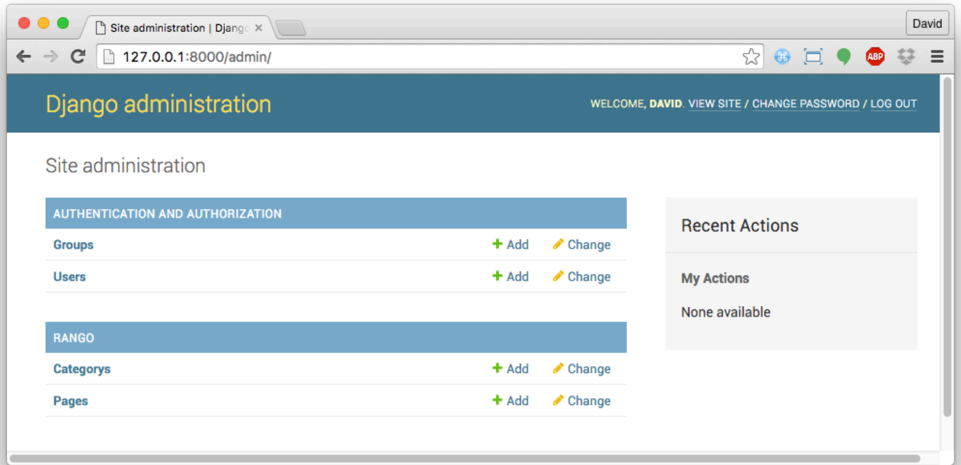


图 5-3：Django 管理界面中显示了 Rango 应用的模型

点击“Rango”区下的 `Categorys` 链接试试。你应该会看到前面在 Django shell 中创建的 Test 分类。

</> 熟悉管理界面

在开发 Rango 应用的过程中我们会经常在管理界面中确认数据有没有正确存储，因此请花点时间熟悉一下管理界面。

把前面创建的 Test 分类删除。稍后会向数据库中填充更多示例数据。

★ 用户管理 ★

用户也在 Django 管理界面中管理，在“Authentication and Authorisation”区。你可以创建、修改和删除用户账户，还可以赋予用户不同的权限等级。

■ 单复数 ■

发现没有，管理界面中有拼写错误（应该是 `Categories`，而不是 `Categorys`）？为了修正这个错误，可以在模型定义中添加嵌套的 `Meta` 类，在里面声明 `verbose_name_plural` 属性。下面是修改后的 `Category` 模型。`Meta` 类有很多作用，详情参见 Django 文档。

```
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)

    class Meta:
        verbose_name_plural = 'Categories'

    def __str__(self):
        return self.name
```

★ 丰富 *admin.py* 的内容 ★

不难看出，*admin.py* 差不多是 Rango 应用中内容最少的模块。其实，在这个模块中能以多种方式定制管理界面。如果感兴趣，请阅读 Django 文档。

5.7 编写一个填充脚本

把测试数据输入数据库是件麻烦事。很多开发者选择随机按键，输入虚假的测试数据，例如 `wTFzmN00bz7`。与其这样，不如编写一个脚本，把真实可信的数据自动填充到数据库中。如此一来，演示或测试应用时，你将看到合理的示例。而且，部署应用或者与同事分享时，你自己和同事无需自己动手输入示例数据。鉴于此，最好编写一个填充脚本。

下面为 Rango 应用编写一个填充脚本。在 Django 项目的根目录中新建一个 Python 文件，命名为 *populate_rango.py*，写入下述代码：

```
import os
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')

import django
django.setup()
from rango.models import Category, Page

def populate():
    # 首先创建一些字典，列出想添加到各分类的网页
    # 然后创建一个嵌套字典，设置各分类
    # 这么做看起来不易理解，但是便于迭代，方便为模型添加数据
```

```python

    python_pages = [
        {"title": "Official Python Tutorial",
         "url":"http://docs.python.org/2/tutorial/"},
        {"title":"How to Think like a Computer Scientist",
         "url":"http://www.greenteapress.com/thinkpython/"},
        {"title":"Learn Python in 10 Minutes",
         "url":"http://www.korokithakis.net/tutorials/python/"} ]

    django_pages = [
        {"title":"Official Django Tutorial",
         "url":"https://docs.djangoproject.com/en/1.9/intro/tutorial01/"},
        {"title":"Django Rocks",
         "url":"http://www.djangorocks.com/"},
        {"title":"How to Tango with Django",
         "url":"http://www.tangowithdjango.com/"} ]

    other_pages = [
        {"title":"Bottle",
         "url":"http://bottlepy.org/docs/dev/"},
        {"title":"Flask",
         "url":"http://flask.pocoo.org"} ]

    cats = {"Python": {"pages": python_pages},
            "Django": {"pages": django_pages},
            "Other Frameworks": {"pages": other_pages} }

    # 如果想添加更多分类或网页，添加到前面的字典中即可

    # 下述代码迭代 cats 字典，添加各分类，并把相关的网页添加到分类中
    # 如果使用的是 Python 2.x，要使用 cats.iteritems() 迭代
    # 迭代字典的正确方式参见
    # http://docs.quantifiedcode.com/python-anti-patterns/readability/

    for cat, cat_data in cats.items():
        c = add_cat(cat)
        for p in cat_data["pages"]:
            add_page(c, p["title"], p["url"])
```

```
51
52     # 打印添加的分类
53     for c in Category.objects.all():
54         for p in Page.objects.filter(category=c):
55             print("- {0} - {1}".format(str(c), str(p)))
56
57 def add_page(cat, title, url, views=0):
58     p = Page.objects.get_or_create(category=cat, title=title)[0]
59     p.url=url
60     p.views=views
61     p.save()
62     return p
63
64 def add_cat(name):
65     c = Category.objects.get_or_create(name=name)[0]
66     c.save()
67     return c
68
69 # 从这开始执行
70 if __name__ == '__main__':
71     print("Starting Rango population script...")
72     populate()
```

■ 充分理解代码 ■

再次说明，不要直接复制粘贴代码，然后就不管了。把代码添加到模块中之后，逐行分析一下，要理解各行的作用。

下面给出说明，希望你能从中学到新知识。

你可能注意到了，上述代码有行号。这表示给出的是整个文件的代码，而不是代码片段。带行号的代码更难复制。

看着代码很多，其实就是对模块后部定义的 `add_page()` 和 `add_cat()` 两个函数的调用。上述代码从模块的底部开始执行，即第 71 和 72 行。这是因为在此之前是函数定义，并未调用，因此也就未执行。解释器遇到第 70 行的 `if __name__ == '__main__'` 时调用 `populate()` 函数。

■ __name__ == '__main__' 是什么意思？ ■

__name__ == '__main__' 是个有用的技巧，既让 Python 模块作为可重用的模块，也让其作为独立的 Python 脚本。可重用的模块是指可以导入其他模块（例如通过 import 语句），而独立的 Python 脚本可以在终端或命令提示符中执行（python module.py）。

因此，if __name__ == '__main__' 条件语句中的代码仅在作为独立的 Python 脚本运行时才执行。如果作为模块导入，那一段代码不执行，因为导入做是为了访问模块中的类或函数。

◆ 导入模型 ◆

导入 Django 模型之前要导入 django，并把环境变量 DJANGO_SETTINGS_MODULE 设为项目的设置文件，然后调用 django.setup()，导入 Django 项目的设置（第 1-6 行）。

如果缺少这重要的一步，导入模型时会抛出异常，这是因为所需的 Django 基础设施未初始化。鉴于此，我们在第 7 行才导入 Category 和 Page 模型。

第 47-50 行的 for 循环负责不断调用 add_cat() 和 add_page() 函数。这两个函数的作用分别是创建分类和网页。populate() 函数会记录创建了哪些分类。我们在第 48 行把分类存储在局部变量 c 中，因为创建 Page 对象时要引用 Category 对象。调用 add_cat() 和 add_page() 函数后，populate() 迭代所有 Category 和相应的 Page 对象，把它们的名称显示在终端中。

★ 创建模型实例 ★

在上述填充脚本中，我们使用便利的 get_or_create() 方法创建模型实例。由于我们不想创建重复的记录，因此使用 get_or_create() 方法检查数据库中有没有要创建的记录；如果没有，创建指定的记录；否则，返回那个模型实例的引用。

这个辅助方法能避免我们编写大量重复的代码。既然有这样的方法存在，何必自己动手检查呢！

get_or_create() 方法返回一个元组 (object, created)。第一个元素是数据库中不存在记录时创建的模型实例引用。这个模型实例使用传给 get_or_create() 方法的参数创建，例如上例中的 category、title、url 和 views。如果数据库中存在相应的记录，get_or_create() 方法返回那个记录。created 是布尔值，get_or_create() 创建模型实例时值为 True。

了解这些之后我们便知道 get_or_create() 后面的 [0] 表示只返回对象引用。与很多其他编

程语言数据结构一样，Python 元组中的元素从零开始编号。

`get_or_create()` 方法的更多信息参见 Django 文档。

保存文件，在终端把当前工作目录切换到 Django 项目的根目录，然后执行 `python populate_rango.py` 命令。你应该会看到类似下面的输出，在不同的电脑中添加分类的顺序可能有所不同。

```
$ python populate_rango.py

Starting Rango population script...
- Python - Official Python Tutorial
- Python - How to Think like a Computer Scientist
- Python - Learn Python in 10 Minutes
- Django - Official Django Tutorial
- Django - Django Rocks
- Django - How to Tango with Django
- Other Frameworks - Bottle
- Other Frameworks - Flask
```

然后，确认填充脚本确实把数据填充进数据库中了。重启 Django 开发服务器，访问管理界面（*http://127.0.0.1:8000/admin/*），检查有没有新的分类和网页出现。点击“Pages”，有没有看到所有网页，如下图所示？

编写填充脚本要花点时间，但从长远来看，最终是能节省时间的。把应用部署到其他地方，只需运行填充脚本就能立即演示，这样多方便。单元测试也能用到填充脚本（参见第 17 章）。

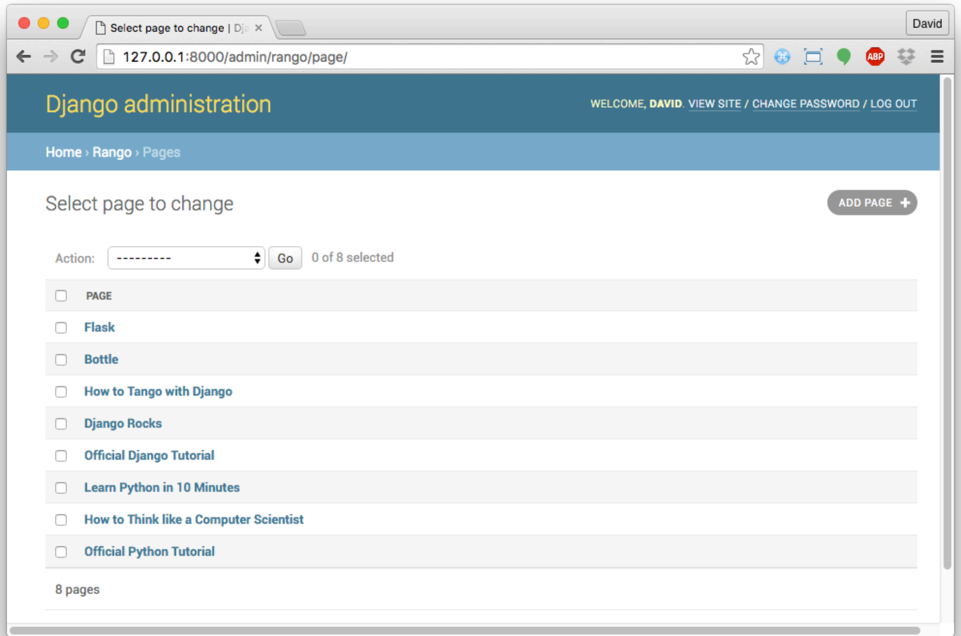


图 5-4：Django 管理页面中显示着填充脚本添加的网页

5.8 基本流程

我们已经学习了 Django ORM 的核心概念，现在是时候总结一下了。为了便于参考，笔者把整个过程分成了几部分。如果以后你想回顾这个过程，可以随时翻阅这一节。

设置数据库

新建项目后应该告诉 Django 你想用什么数据库（*settings.py* 模块中的 `DATABASES` 设置）。此外，还可以在 *admin.py* 模块中注册模型，以便在管理界面中管理。

添加模型

添加模型的过程可以分为以下 5 步。

1. 首先在 Django 应用的 *models.py* 文件中定义模型。
2. 更新 *admin.py*，注册新增的模型。
3. 生成迁移：`python manage.py makemigrations <app_name>`。
4. 运行迁移：`python manage.py migrate`。在数据库中创建模型所需的表和字段。

❺ 创建或编辑填充脚本。

有时你可能想删除数据库，重头再来。具体步骤如下。注意，本书使用的是 SQLite 数据库，此外 Django 还支持其他数据库引擎。

❶ 如果 Django 开发服务器正在运行，停止。

❷ 如果使用的是 SQLite 数据库，删除 Django 项目根目录中的 *db.sqlite3* 文件。这个文件与 *manage.py* 脚本位于同一级目录中。

❸ 如果修改过应用的模型，执行 `python manage.py makemigrations <app_name>` 命令，记得把 `<app_name>` 替换成 Django 应用的名称（例如 `rango`。）如果未修改模型，跳过这一步。

❹ 执行 `python manage.py migrate` 命令，新建数据库文件（使用 SQLite 的话），并创建数据库表。

❺ 执行 `python manage.py createsuperuser` 命令，创建一个超级用户。

❻ 最后，运行填充脚本，在新数据库中插入可信的测试数据。

</> 练习

请完成以下练习，巩固本章所学的知识。再次提醒，后续章节建立在这些练习之上，请务必完成。如果卡住了，请看后面的提示。

- 更新 `Category` 模型，加上 `views` 和 `likes` 字段，二者的默认值均是零。
- 创建迁移，然后执行，提交此次改动。
- 更新填充脚本，把“Python”分类的查看次数设为 128、点赞次数设为 64，把“Django”分类的查看次数设为 64、点赞次数设为 32，把“Other Frameworks”分类的查看次数设为 32、点赞次数设为 16。
- 删除数据库，然后重新创建，再使用填充脚本填充数据。
- 阅读 Django 官方教程的第二部分和第七部分。这两部分能巩固本章所学的数据库处理知识，还涉及一些定制 Django 管理界面的技术。
- 定制管理界面，访问 `Page` 模型时显示网页的分类、名称和 URL，就像图 5-5 那样。你要做完前一题才知道怎么解答这一题。

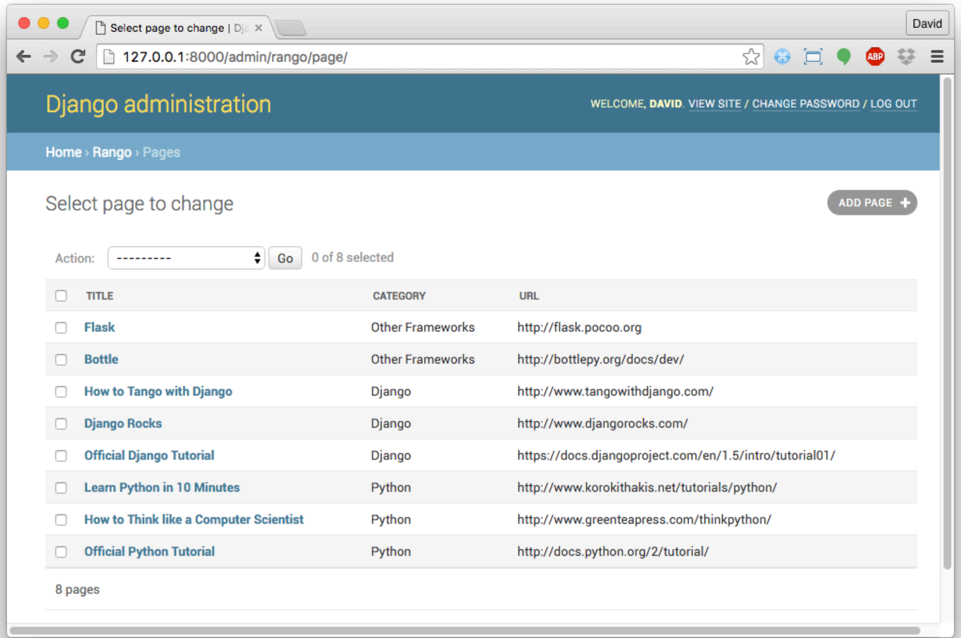


图 5-5：更新后的管理界面，Page 模型页面显示有网页的分类和 URL

★ 提示 ★

如果你做不出上述练习，希望下面的提示能给你一点启发。

- 修改 `Category` 模型，添加两个 `IntegerField`：`views` 和 `likes`。
- 修改填充脚本中的 `add_cat()` 函数，让它能处理 `Category` 模型新增的 `views` 和 `likes` 字段。
    - 除了 `name` 参数之外，为 `add_cat()` 函数增加两个参数，以便传入 `views` 和 `likes` 的值。
    - 在 `add_cat()` 函数中使用这两个参数设置 `Category` 模型实例的 `views` 和 `likes` 字段。在填充脚本中，分类对象赋值给变量 `c`，因此可以通过 `c.likes` 访问 `likes` 字段。别忘了调用 `save()`，保存实例。
    - 然后更新填充脚本中 `populate()` 函数里的 `cats` 字典。这个字典中的元素，键是分类的名称，值也是一个字典，包含分类的信息。要修改的是这个内部的字典，为各分类添加 `views` 和 `likes`。

- 最后，修改调用 `add_cat()` 函数的方式。现在要传入三个参数（`name`、`views` 和 `likes`），而目前只传入了 `name`。要在函数调用中添加那两个参数。如果不知道如何使用 `for` 循环迭代字典，请阅读这个在线 Python 教程。读完之后你便知道如何访问字典中的 `views` 和 `likes` 值。

❑ 更新填充脚本之后，定制管理界面。要编辑 *rango/admin.py* 文件，定义 `PageAdmin` 类，继承自 `admin.ModelAdmin`。

- 在新增的 `PageAdmin` 类中添加 `list_display = ('title', 'category', 'url')`。
- 然后注册 `PageAdmin` 类。要修改 Rango 应用的 *admin.py* 文件，把 `admin.site.register(Page)` 改成 `admin.site.register(Page, PageAdmin)`。

★ 测试 ★

笔者编写了几个测试，检查你有没有做完练习。从本书的 GitHub 仓库中下载 *tests.py* 脚本，保存到 Rango 应用目录中。

然后在终端或命令提示符中执行下述命令，运行测试：

```
$ python manage.py test rango
```

如果你对自动化测试感兴趣，可以翻到第 17 章。那一章将介绍一些基本的测试知识，以便自动检查代码的完整性。

6 模型、模板和视图

现在，我们定义了两个模型，还在数据库中填充了一些示例数据，接下来可以连接模型、视图和模板，伺服动态内容了。本章说明在首页显示分类的过程，并为各分类创建专属页面，显示分类下的链接。

6.1 创建数据驱动页面的流程

在 Django 中创建数据驱动页面主要分为 5 步：

1. 在 *views.py* 文件中导入想使用的模型。
2. 在视图函数中查询模型，获取想呈现的数据。
3. 把从模型获取的数据传给模板上下文。
4. 创建或修改模板，显示上下文中的数据。
5. 把 URL 映射到视图上（如果还未做的话）。

以上就是在 Django 框架中把模型、视图和模板连接在一起的步骤。

6.2 在首页显示分类

客户对主页的要求之一是显示最受欢迎的 5 个分类。下面根据上述步骤实现这一要求。

导入所需的模型

首先完成第一步。打开 *rango/views.py* 文件，在顶部其他导入语句之后从 Rango 应用的 *models.py* 文件中导入 `Category` 模型：

```
# 导入 Category 模型
from rango.models import Category
```

修改 index 视图

下面完成第 2 步和第 3 步。我们要修改主页的视图函数，即 index()。根据下述代码修改。

```
def index(request):
    # 查询数据库，获取目前存储的所有分类
    # 按点赞次数倒序排列分类
    # 获取前 5 个分类（如果分类数少于 5 个，那就获取全部）
    # 把分类列表放入 context_dict 字典
    # 稍后传给模板引擎

    category_list = Category.objects.order_by('-likes')[:5]
    context_dict = {'categories': category_list}

    # 渲染响应，发给客户端
    return render(request, 'rango/index.html', context_dict)
```

这里的 Category.objects.order_by('-likes')[:5] 从 Category 模型中查询最受欢迎的前 5 个分类。order_by() 方法的作用是排序，这里我们根据 likes 字段的值倒序排列。-likes 中的 - 号表示倒序（如果没有 - 号，返回的结果是升序排列的）。因为我们想获得一个分类对象列表，所以使用 Python 的列表运算符从列表中获取前 5 个对象（[:5]），返回一个 Category 对象子集。

查询结束后，把列表的引用（category_list 变量）传给 context_dict 字典。最后把这个字典作为模板上下文传给 render() 函数。

◆ 警告 ◆

在此之前务必做完前一章的练习，即为 Category 模型添加 likes 字段。

修改 index 模板

更新视图之后，接下来要做第 4 步，更新项目根目录中 *templates* 目录里的 *rango/index.html* 模板。根据下述代码片段修改模板中的 HTML。

```
<!DOCTYPE html>
```

```
{% load staticfiles %}
<html>
<head>
    <title>Rango</title>
</head>

<body>
    <h1>Rango says...</h1>
    <div>hey there partner!</div>

    <div>
    {% if categories %}
    <ul>
        {% for category in categories %}
            <li>{{ category.name }}</li>
        {% endfor %}
    </ul>
    {% else %}
        <strong>There are no categories present.</strong>
    {% endif %}
    </div>

    <div>
        <a href="/rango/about/">About Rango</a><br />
        <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    </div>
</body>
</html>
```

这里，我们使用 Django 模板语言提供的 if 和 for 控制语句呈现数据。在页面的 <body> 元素中，我们判断 categories（包含分类列表的上下文变量）中有没有分类（{% if categories %}）。

如果有分类，构建一个 HTML 无序列表（ 标签）。for 循环迭代分类列表（{% for category in categories %}），在列表项目（ 标签）中输出各分类的名称（{{ category.name }}）。

如果没有分类，显示一个消息，指明没有分类。

从上述代码片段可以看出，Django 模板语言中的命令放在 {% 和 %} 之间，而变量放在 {{ 和 }} 之间。

现在访问 Rango 的主页应该会看到页面标题下方显示着分类列表，如图 6-1 所示。

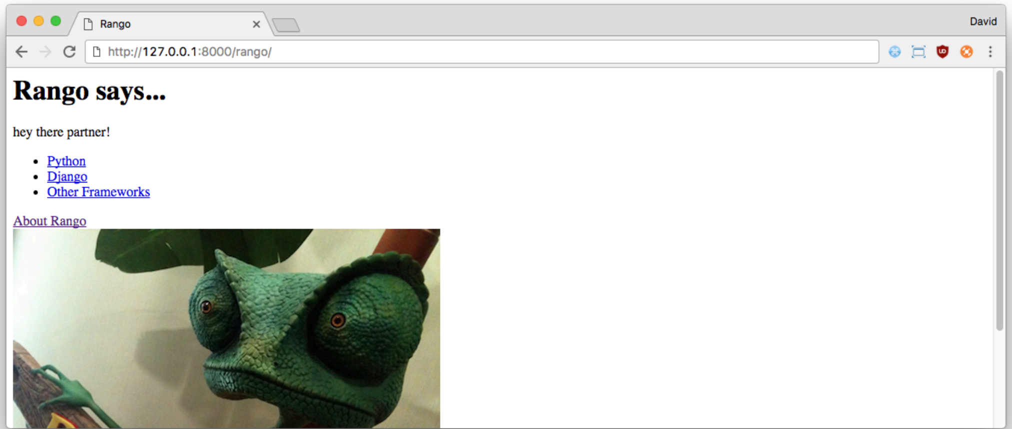


图 6-1：Rango 首页显示动态生成的分类列表

6.3 创建详情页面

根据设计要求，每个分类还有对应的详情页面。这里有些挑战要克服。我们要定义一个参数化视图，而且 URL 模式中要编码分类名称。

URL 设计和映射

先解决 URL 问题。我们可以在 URL 中使用分类的唯一 ID，例如 */rango/category/1/* 或 */rango/category/2/*，其中的数字 1 和 2 是分类的唯一 ID。可是从 ID 上看不出到底是哪个分类。

更好的方法是在 URL 中使用分类的名称。例如，可以通过 URL */rango/category/python/* 访问与 Python 相关的网页列表。这样的 URL 简单、可读性高，而且具有一定的语义。如果采用这种形式，还要处理有多个单词的分类名，例如“Other Frameworks”。

★ 简洁的 URL ★

使用简洁且可读性高的 URL 是 Web 设计的重要一环。详情参见维基百科。

为此，我们要使用 Django 提供的 `slugify` 函数。

为分类添加 slug 字段

为了得到可读性高的 URL，我们要为 Category 模型添加一个别名（slug）字段。然后使用 Django 提供的 slugify 函数把空白替换为连字符，例如 "how do i create a slug in django" 将变成 "how-do-i-create-a-slug-in-django"。

> **◆ 不安全的 URL ◆**
>
> URL 中虽然可以有空格，但是却会导致安全问题。详情参见因特网工程任务组（Internet Engineering Task Force）发布的 URL RFC。

此外，还要覆盖 Category 模型的 save() 方法，调用 slugify() 函数更新 slug 字段。注意，只要分类名称变了，别名就随之改变。参照下述代码片段更新模型，别忘了导入 slugify() 函数。

```
from django.template.defaultfilters import slugify
...
class Category(models.Model):
    name = models.CharField(max_length=128, unique=True)
    views = models.IntegerField(default=0)
    likes = models.IntegerField(default=0)
    slug = models.SlugField()

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(Category, self).save(*args, **kwargs)

    class Meta:
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name
```

更新模型之后，接下来要把变动应用到数据库上。然而，数据库中已经有数据了，因此我们必须考虑改动产生的影响。其实我们想做的很简单，就是从分类名称中得到别名（此项操作在初次保存记录时执行）。通过迁移工具能把 slug 字段添加到数据库中，而且可以为该字段指定默认值。可是，每个分类的别名应该是不同的。因此，我们将先执行迁移，然后重新运行填充脚本。之所以这么做，是因为填充脚本会在分类上调用 save() 方法，从而触发上面实现的 save() 方法，更

新各分类的别名。

执行下述命令，执行迁移：

```
$ python manage.py makemigrations rango
$ python manage.py migrate
```

我们没有为 slug 字段指定默认值，而且数据库中有数据，因此 migrate 命令会给你两个选择。选择提供默认值的选项，输入一个空字符串（两个引号，即 ''）。然后运行填充脚本，更新 slug 字段。

```
$ python populate_rango.py
```

执行 python manage.py runserver 命令，启动 Django 开发服务器，在管理界面中查看模型中的数据。

如果此时通过管理界面添加分类，可能会遇到一两个问题。

1. 假设要添加的分类名是“Python User Groups”。保存时 Django 会阻止你，让你填写别名。虽然可以自己动手输入“python-user-groups”，但是这样容易出错。如果能自动生成别名就好了。
2. 如果有个分类名为“Django”，还有个名为“django”，又会遇到一个问题。因为 slugify() 函数生成的别名是小写形式，所以无法区分别名“django”对应于哪个分类。

第一个问题的解决方法有两个。其一，更新模型，把 slug 字段设为允许空值：

```
slug = models.SlugField(blank=True)
```

其二，定制管理界面，在输入分类名称时自动填写别名。请参照下述代码更新 *rango/admin.py*。

```
from django.contrib import admin
from rango.models import Category, Page
...
# 添加这个类，定制管理界面
class CategoryAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug':('name',)}

# 注册定制界面的类
admin.site.register(Category, CategoryAdmin)
...
```

在管理界面中添加一个分类试试。

第二个问题也不难解决，只需把 `slug` 字段设为唯一的。为 `slug` 字段添加约束：

```
slug = models.SlugField(unique=True)
```

添加这个约束之后便可以通过别名唯一标识分类。你可能想在之前就添加唯一性约束，但是这样一来执行迁移时把别名都设为空字符串就会遇到问题，因为违背了唯一性约束。此外也可以删除数据库，然后重建，但这并不适合所有情况。

◆ 迁移混乱 ◆

> 事先最好规划好数据库，尽量不修改。填充脚本的作用是便于删除数据库后重建。
>
> 有时，删除数据库后重建比找出问题后再设法解决要方便。你可以练习编写一个脚本，输出数据库中的数据。这样每次修改数据库都可以把数据输出到一个文件中保存起来，供以后查看。

创建分类页面的步骤

为了实现可通过 */rango/category/<category-name-slug>/* 访问的分类页面，我们要做几处修改。基本步骤如下：

1. 把 `Page` 模型导入 *rango/views.py* 模块。
2. 在 *rango/views.py* 模块中定义一个新视图，命名为 `show_category()`。这个视图有个额外的参数，`category_name_slug`，用于传入编码后的分类名称。为了编码和解码 `category_name_slug`，要定义两个辅助函数。
3. 创建一个模板，*templates/rango/category.html*。
4. 更新 *rango/urls.py*. 中的 `urlpatterns`，把这个新视图映射到 URL 模式上。

此外还要更新 `index()` 视图和 *index.html* 模板，添加指向分类页面的链接。

分类视图

在 *rango/views.py* 中，首先导入 `Page` 模型，即把下述导入语句添加到文件顶部：

```python
from rango.models import Page
```

然后定义视图 show_category()。

```python
def show_category(request, category_name_slug):
    # 创建上下文字典，稍后传给模板渲染引擎
    context_dict = {}

    try:
        # 能通过传入的分类别名找到对应的分类吗?
        # 如果找不到，.get() 方法抛出 DoesNotExist 异常
        # 因此 .get() 方法返回一个模型实例或抛出异常
        category = Category.objects.get(slug=category_name_slug)

        # 检索关联的所有网页
        # 注意，filter() 返回一个网页对象列表或空列表
        pages = Page.objects.filter(category=category)

        # 把得到的列表赋值给模板上下文中名为 pages 的键
        context_dict['pages'] = pages
        # 也把从数据库中获取的 category 对象添加到上下文字典中
        # 我们将在模板中通过这个变量确认分类是否存在
        context_dict['category'] = category
    except Category.DoesNotExist:
        # 没找到指定的分类时执行这里
        # 什么也不做
        # 模板会显示消息，指明分类不存在
        context_dict['category'] = None
        context_dict['pages'] = None

    # 渲染响应，返回给客户端
    return render(request, 'rango/category.html', context_dict)
```

这个视图的基本步骤与 index() 视图一样。首先定义上下文字典，然后尝试从模型中提取数据，并把相关数据添加到上下文字典中。我们通过传给 show_category() 视图函数的 category_name_slug 参数确认要查看的是哪个分类。如果通过别名找到了分类，获取与之关联的网页，并将其添加到上下文字典 context_dict 中。

分类模板

下面为这个新视图创建模板。在 *<workspace>/tango_with_django_project/templates/rango/* 目录中新建 *category.html* 文件，写入下述代码。

```
<!DOCTYPE html>
<html>
<head>
    <title>Rango</title>
</head>
<body>
    <div>
    {% if category %}
        <h1>{{ category.name }}</h1>
        {% if pages %}
            <ul>
            {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
            {% endfor %}
            </ul>
        {% else %}
            <strong>No pages currently in category.</strong>
        {% endif %}
    {% else %}
        The specified category does not exist!
    {% endif %}
    </div>
</body>
</html>
```

上述 HTML 代码再次展示了如何通过 `{{ }}` 标签使用模板上下文中的数据。我们访问了 `category` 和 `pages` 对象，以及它们的字段，例如 `category.name` 和 `page.url`。

如果 `category` 存在，再检查分类下有没有网页。如果有，使用模板标签 `{% for page in pages %}` 迭代网页，显示 `pages` 列表中各网页的 `title` 和 `url` 属性。网页的信息在一个 HTML 无序列表（`<ul>` 标签）中显示。如果你不熟悉 HTML，可以查看 W3Schools.com 网站中的 HTML 教程，学习不同的标签。

■ 条件判断模板标签 ■

使用 Django 模板的条件标签 {% if %} 判断上下文中有没有某个对象十分方便。为了避免出错，应该检查对象是否存在。

在模板中检查条件（例如上例中的 {% if category %}）也更符合语义。条件判断的结果直接影响呈献给用户的页面。记住，Django 应用表现层的逻辑应该封装在模板中。

带参数的 URL 映射

下面看看 category_name_slug 参数的值是如何传给 show_category() 视图函数的。打开 Rango 应用的 *urls.py* 文件，把 urlpatterns 改成下面这样：

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^about/$', views.about, name='about'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

我们添加了一个相当复杂的 URL 模式，当 URL 匹配 r'^category/(?P<category_name_slug>[\w\-]+)/$' 时调用 views.show_category() 函数。

这里有两点要注意。首先，URL 模式中有个参数，即 <category_name_slug>，在视图中可以访问。声明带参数的 URL 时，要确保对应的视图中有那个参数。其次，正则表达式 [\w\-]+ 匹配连续的数字字母（即 a-z、A-Z 或 0-9，正则表达式中的 \w）和连字符（正则表达式中的 \-），而且可以匹配任意个（正则表达式中的 [ ]+）。

新增的 URL 模型匹配 rango/category/ 和末尾的 / 之间的数字字母和连字符序列。匹配的序列存储在参数 category_name_slug 中，传给 views.show_category() 函数。例如，对 *rango/category/python-books/* 这个 URL 来说，category_name_slug 参数的值是 python-books。然而，如果 URL 是 *rango/category/££££-$/*，那么 rango/category/ 和末尾的 / 之间的字符与正则表达式不匹配，此时将得到“404 not found”错误，因为没有与之匹配的 URL 模式。

Django 应用中的视图函数必须至少有一个参数。这个参数通常命名为 request，通过它获取与 HTTP 请求有关的信息。如果 URL 中带有参数，必须为对应的视图函数声明额外的具名参数。鉴于此，show_category() 视图才定义为 def show_category(request, category_name_slug)。

★ 正则表达式恐惧症 ★

有些人遇到问题时会想，“我知道，使用正则表达式嘛！”现在他们将面临两个问题。

—— Jamie Zawinski

正则表达式看似令人恐惧，晦涩难懂，但是网上有大量资源能助你一臂之力。遇到问题时，可以使用这个速查表。

修改 index 模板

新视图可用了，但是还有一件事要做。我们要修改首页的模板，为列出的分类添加链接，指向分类页面。更新 *index.html* 模板，加入指向分类页面的链接。

```
<!DOCTYPE html>
{% load staticfiles %}
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Rango says...</h1>

        <div>
            hey there partner!
        </div>

        <div>
        {% if categories %}
        <ul>
            {% for category in categories %}
            <!-- 修改下面这一行，添加链接 -->
            <li>
            <a href="/rango/category/{{ category.slug }}">{{ category.name }}</a>
            </li>
            {% endfor %}
        </ul>
        {% else %}
            <strong>There are no categories present.</strong>
```

```
            {% endif %}
            </div>

            <div>
                <a href="/rango/about/">About Rango</a><br />
                <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
            </div>
        </body>
    </html>
```

这里也是使用 HTML `<ul>` 标签定义一个无序列表，里面有一系列列表项目（`<li>`），其中有一个 HTML 超链接（`<a>`）。超链接有个 `href` 属性，其值为 `/rango/category/{{ category.slug }}`。例如，“Python Books”分类对应的 URL 是 `/rango/category/python-books/`。

检验结果

访问 Rango 首页，检查一下劳动成果。你应该会看到首页最多显示 5 个分类，而且都带超链接。点击“Django”会转到 Django 分类的页面，如图 6-2 所示。如果你看到“Official Django Tutorial”链接，说明你做的没错。

访问不存在的分类情况如何呢？访问一个不存在的分类试试，例如直接在浏览器的地址栏中输入 */rango/category/computers/*。你应该会看到一个消息，提示要查看的分类不存在。



图 6-2：Django 分类页面显示的链接。注意，鼠标悬停在第一个链接上，在 Google Chrome 浏览器窗口的底部能看到对应的 URL。

</> 练习

请完成以下练习，巩固本章所学的知识。

- ❏ 更新填充脚本，为各网页添加访问次数（`views` 字段）。
- ❏ 修改首页，加上访问次数最多的 5 个网页。
- ❏ 为两块信息添加标题，分别为“Most Liked Categories”和“Most Viewed Pages”。
- ❏ 在分类页面添加回到首页的链接。
- ❏ 阅读 Django 官方教程第三部分，巩固本章所学。

★ 提示 ★

- ❏ 更新填充脚本的基本步骤与前一章的练习一样。各网页的数据结构，以及处理它们的代码都要更新。
  - 更新三个分类中网页的数据结构。目前，每个网页有 `title` 和 `url` 字段。现在，加上访问次数，即 `views` 字段。
  - 填充脚本中的 `add_page()` 函数允许传入访问次数吗？要做什么修改吗？
  - 最后，更新调用 `add_page()` 函数那行代码。如果数据结构中的访问次数用 `views` 键表示，引用网页的变量为 `p`，应该如何把访问次数传给 `add_page()` 函数呢？
- ❏ 记得重新运行填充脚本，更新网页的访问次数。
  - 更新 `index` 视图和 *index.html* 模板，在首页显示访问次数最多的网页。
  - 现在不访问 `Category` 模型了，要通过 `Page` 模型获取访问次数最多的网页。
  - 记得把网页列表添加到上下文中。
  - 如果不知道怎么编写模板，可以参考 *category.html* 模板。二者基本是一致的。

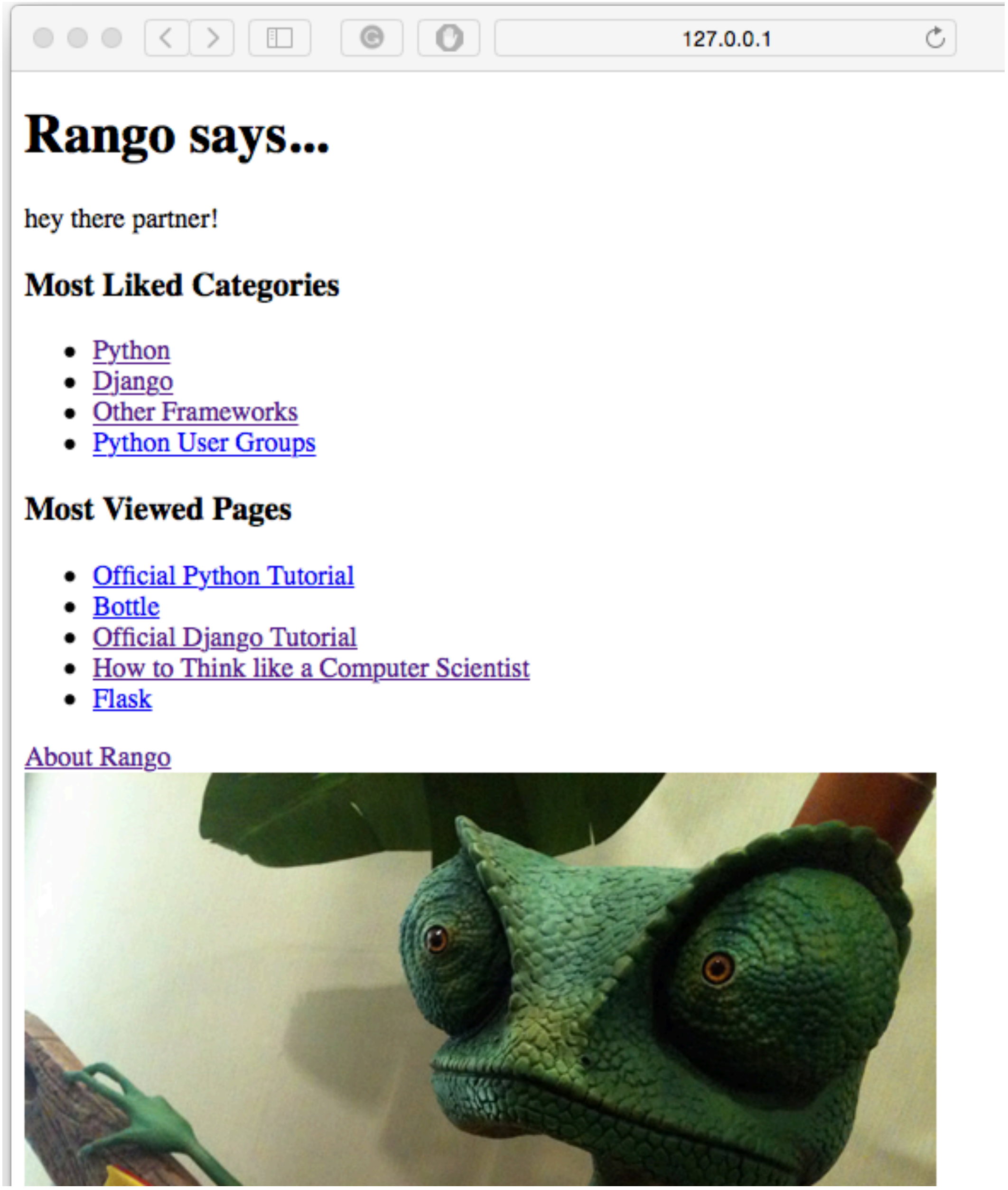


图 6-3：做完练习后的首页，显示着最受欢迎的分类和访问次数最多的网页

★ 深入了解模型 ★

如果想深入了解模型，可以阅读下面两篇博客文章。

1. Kostantin Moiseenko 写的 Django 模型最佳实践。这篇文章给出了很多处理模型的技巧。

❷ Robert Roskam 写的 Django 模型去除重复指南。这篇文章介绍如何通过类的 `property` 方法减少访问关联模型的代码量。

7 表单

本章说明如何通过 Web 表单捕获数据。Django 提供的表单处理功能简单明了，根据 Django 文档，通过这个功能可以做到：

1. 自动生成 HTML 表单的小组件（例如文本字段或日期选择器）；
2. 检查提交的数据是否满足验证规则；
3. 遇到验证错误时重新显示表单；
4. 把提交的表单数据转换成相应的 Python 数据类型。

使用 Django 表单功能的一大优势是能节省大量时间，免去自己动手创建 HTML 表单的繁琐过程。

7.1 基本流程

创建表单处理用户输入的基本步骤如下。

1. 在 Django 应用的目录中创建 *forms.py* 文件，存放表单相关的类。
2. 为想使用表单处理的模型定义 `ModelForm` 类。
3. 根据需要定制表单。
4. 创建或更新视图，处理表单：
    - 显示表单
    - 保存表单数据
    - 用户输入错误的数据时（或者根本没输入数据时）报错

❺ 创建或更新模板，显示表单。

❻ 添加 URL 模式，映射到视图上（如果视图是新的）。

这个流程比以往复杂一些，而且相应的视图要复杂得多。不过，多做几次便能掌握。

7.2 网页和分类表单

本节实现的功能是为了让用户通过表单向数据库中添加分类和网页。

首先，在 Rango 应用的目录中新建一个文件，命名为 *forms.py*。这一步不是必须的（可以放在 *models.py* 文件中），但是把表单相关的代码放在单独的文件中有利于保持代码基整洁。

定义 `ModelForm` 的子类

在 *forms.py* 模块中，我们将定义几个继承自 `ModelForm` 的类。`ModelForm` 是 Django 提供的辅助类，简化了为模型创建表单的过程。Rango 应用现在有两个模型（`Category` 和 `Page`），我们将分别为二者定义 `ModelForm` 的子类。

在 *rango/forms.py* 文件中添加下述代码。

```
from django import forms
from rango.models import Page, Category

class CategoryForm(forms.ModelForm):
    name = forms.CharField(max_length=128,
                           help_text="Please enter the category name.")
    views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
    likes = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
    slug = forms.CharField(widget=forms.HiddenInput(), required=False)

    # 嵌套的类，为表单提供额外信息
    class Meta:
        # 把这个 ModelForm 与一个模型连接起来
        model = Category
        fields = ('name',)

class PageForm(forms.ModelForm):
```

```
    title = forms.CharField(max_length=128,
                            help_text="Please enter the title of the page.")
    url = forms.URLField(max_length=200,
                         help_text="Please enter the URL of the page.")
    views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)

    class Meta:
        # 把这个 ModelForm 与一个模型连接起来
        model = Page

        # 想在表单中放哪些字段?
        # 有时不需要全部字段
        # 有些字段接受空值，因此可能无需显示
        # 这里我们想隐藏外键字段
        # 为此，可以排除 category 字段
        exclude = ('category',)
        # 也可以直接指定想显示的字段（不含 category 字段）
        # fields = ('title', 'url', 'views')
```

我们要通过 `fields` 指定表单中包含哪些字段，或者通过 `exclude` 指定排除哪些字段。

为了定制得到的表单，Django 提供了多种方式。在上述代码中，我们指明了使用什么小组件（widget）显示各个字段。例如，在 `PageForm` 类中，`title` 字段用的是 `forms.CharField`，`url` 字段用的是 `forms.URLField`。这两种小组件都用于输入文本。注意我们为字段设定的 `max_length` 参数，其值与底层数据模型中设定的长度限制一样。具体限制参见第 5 章，或者看一下 Rango 应用的 *models.py* 文件。

此外，两个表单中都有 `IntegerField`，用于呈现查看（访问）次数和点赞次数。注意，我们在参数中设定了 `widget=forms.HiddenInput()`，这是为了隐藏小组件。另外，我们还设定了 `initial=0`，把值设为零。这是把字段的默认值设为零的一种方式。因为字段是隐藏的，用户无法为其输入值。

然而，如 `PageForm` 所示，尽管有个字段是隐藏的，但是依然要将其包含在表单的字段中。如果 `fields` 中没有 `views`，表单中就没有那个字段（虽然声明了），因此那个字段的初始值也就不会是零。在某些模型中，这样做可能导致报错。如果在模型中为字段设定了 `default=0`，那么便可以交由模型自动使用默认值填充字段，从而避免 `not null` 错误。此时，表单中无需包含那个字段。`CategoryForm` 中包含 `slug` 字段，但是没有为其指定初始值或默认值，而是隐藏了，并且指明这个字段不是必须的。这是因为模型在调用 `save()` 方法保存时会生成这个字段的值。综上，定义模型和表单时一定要小心，确保提供创建模型实例所需的全部数据。

除了 `CharField` 和 `IntegerField` 之外，Django 还提供了很多可用的小组件，例如 `EmailField`（电子邮件地址输入框）、`ChoiceField`（单选按钮）和 `DateField`（日期/时间输入框）。

对 `ModelForm` 的子类来说，最重要的一点或许是定义表单对应的模型。这个信息通过嵌套的 `Meta` 类提供，把 `model` 属性设为目标模型。例如，`CategoryForm` 类设定的目标模型是 `Category`。这一步十分重要，有了这个信息 Django 才能根据模型创建表单，并在提交表单后显示可能出现的错误。

我们还通过 `Meta` 类的 `fields` 属性（值为一个元组）设定表单中包含哪些字段。

★ 进一步了解表单 ★

> 如果想进一步了解不同的小组件，以及定制表单的方法，请阅读 Django 文档。

编写添加分类视图

定义好 `CategoryForm` 类之后，接下来要编写一个视图，用于显示表单及处理提交的数据。在 *rango/views.py* 文件中添加下述代码。

```
# 在文件顶部添加这个导入语句
from rango.forms import CategoryForm
...
def add_category(request):
    form = CategoryForm()

    # 是 HTTP POST 请求吗?
    if request.method == 'POST':
        form = CategoryForm(request.POST)

        # 表单数据有效吗?
        if form.is_valid():
            # 把新分类存入数据库
            form.save(commit=True)
            # 保存新分类后可以显示一个确认消息
            # 不过既然最受欢迎的分类在首页
            # 那就把用户带到首页吧
            return index(request)
        else:
```

```
            # 表单数据有错误
            # 直接在终端里打印出来
            print(form.errors)

    # 处理有效数据和无效数据之后
    # 渲染表单，并显示可能出现的错误消息
    return render(request, 'rango/add_category.html', {'form': form})
```

`add_category()` 视图展示了处理表单的关键步骤。首先，创建一个 `CategoryForm` 实例，然后检查是不是 HTTP POST 请求，即是不是通过表单提交的数据？相同的 URL 也可以处理 GET 请求。`add_category()` 视图能处理不同的情况：

- 显示一个空表单，让用户添加分类；
- 把用户提交的数据存入相应的模型，然后渲染 Rango 应用首页；
- 如有错误，再次显示表单，并把错误消息一并显示出来。

■ GET vs. POST? ■

GET 和 POST 是两种 HTTP 请求类型，二者之间的区别如下：

- HTTP GET 请求获取指定资源的表述。即 HTTP GET 请求用于获取特定的资源，例如一个网页、一张图像或一个文件。
- HTTP POST 请求提交客户端中的数据，供服务器处理。提交 HTML 表单发送的就是这种请求。HTTP POST 请求的最终结果可能是在服务器中新建一个资源（并存入数据库）。新建的资源可通过 HTTP GET 请求访问。

详情参见 w3schools 网站中对二者的比较。

Django 的表单处理机制负责处理通过 HTTP POST 请求提交的数据，一切顺利时，把数据存入相应的模型，否则生成各字段的出错消息。也就是说，如果提交的表单数据违背了数据库的引用完整性，Django 会拒绝保存数据。例如，如果不为 `category` 字段提供值就会出错，因为这个字段不能为空。

从调用 `render()` 函数那行可以看出，我们使用的是一个新模板，名为 *add_category.html*。这个页面和表单的 HTML 代码及相关的 Django 模板代码都在这个模板中。

创建添加分类页面的模板

新建 *templates/rango/add_category.html* 文件，写入下述 HTML 标记和 Django 模板代码。

```
<!DOCTYPE html>
<html>
    <head>
        <title>Rango</title>
    </head>

    <body>
        <h1>Add a Category</h1>
        <div>
            <form id="category_form" method="post" action="/rango/add_category/">
                {% csrf_token %}
                {% for hidden in form.hidden_fields %}
                    {{ hidden }}
                {% endfor %}
                {% for field in form.visible_fields %}
                    {{ field.errors }}
                    {{ field.help_text }}
                    {{ field }}
                {% endfor %}
                <input type="submit" name="submit" value="Create Category" />
            </form>
        </div>
    </body>
</html>
```

这个 HTML 页面的 `<body>` 中有个 `<form>` 元素。通过 `<form>` 元素的属性可以看出，这个表单捕获的数据通过 HTTP POST 请求发给 URL */rango/add_category/*（`method` 属性的值不区分大小写，因此 `POST` 或 `post` 都行）。这个表单中有两个循环：

- 一个显示表单的隐藏字段
- 一个显示表单的可见字段

可见字段，即会显示出来的字段，由 `Meta` 类的 `fields` 属性控制。这两个循环负责生成各表单元素的 HTML 标记。可见字段中还加上了可能出现的错误，以及辅助文本，提示用户应该输入什

么。

■ 隐藏字段 ■

因为 HTTP 是无状态的协议，所以除了可见字段之外，还要加上隐藏字段。由于两个 HTTP 请求之间无法持久保持状态，因此 Web 应用的某些功能难以实现。为了解决这一难题，人们发明了隐藏字段，以便通过 HTML 表单发送对客户端来说重要的信息（在渲染的页面中看不到），在用户提交表单时发给服务器。

◆ 跨站请求伪造令牌 ◆

请特别注意上述代码片段中的 `{% csrf_token %}`。这是*跨站请求伪造*（Cross-Site Request Forgery，CSRF）令牌，用于增强提交表单后发送的 HTTP POST 请求的安全性。Django 框架要求表单中必须有 CSRF 令牌，否则用户提交表单后可能遇到错误。详情参见 Django 文档（Django 1.10 的文档）。

映射添加分类视图

接下来要把 `add_category()` 视图映射到一个 URL 上。在模板中，表单的 `action` 属性使用的 URL 是 */rango/add_category/*。现在要把这个 URL 映射到视图上。打开 *rango/urls.py* 文件，修改 `urlpatterns`：

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),
    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category, name='show_category'),
]
```

这个 URL 模式的位置没什么关系。不过，请阅读 Django 文档，了解 Django 是如何处理请求的。添加分类页面的 URL 是 */rango/add_category/*。

修改首页视图

最后，在首页添加一个链接，方便用户添加分类。打开 *rango/index.html* 模板，在“关于”页面链接所在的 `<div>` 元素中添加下述 HTML 超链接：

```
<a href="/rango/add_category/">Add a New Category</a><br />
```

检验结果

现在检验一下结果。启动或重启 Django 开发服务器，打开浏览器，访问 *http://127.0.0.1:8000/rango/*。点击新加的那个链接，跳转到添加分类页面，添加一个分类试试。下图是首页和添加分类页面的截图。

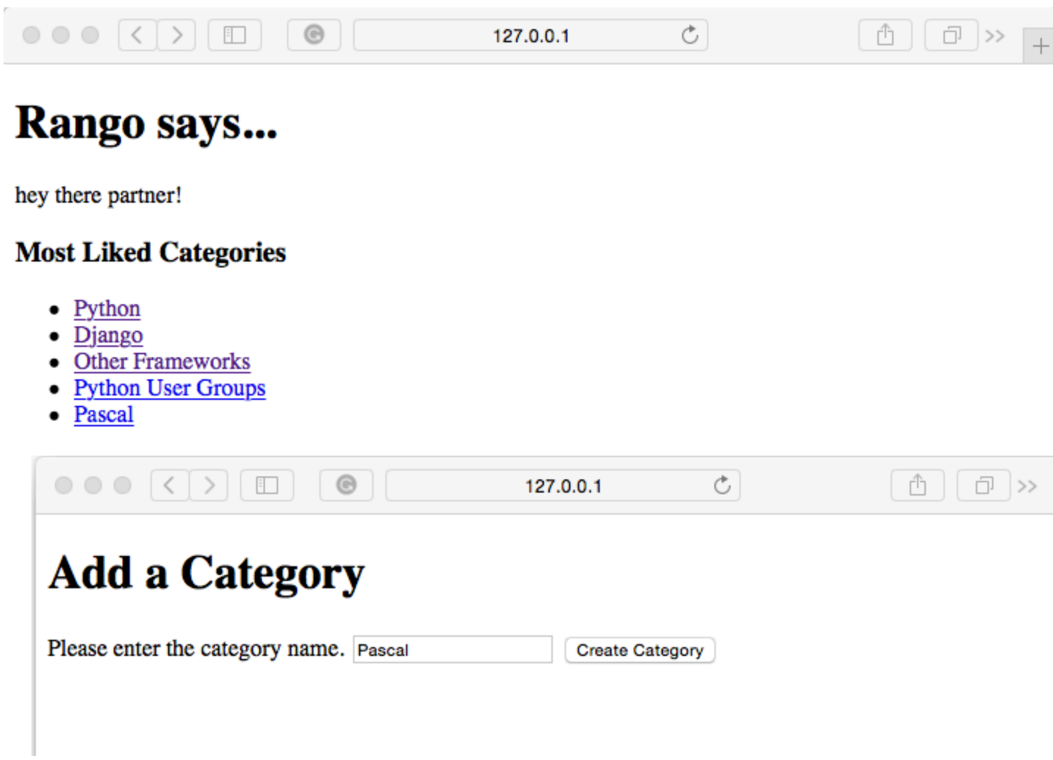


图 7-1：通过表单添加一个分类

■ 分类不见了？ ■

你添加的分类可能不会出现在首页，这是因为首页只显示最受欢迎的前 5 个分类。登录进入管理界面可以看到全部分类。

确认分类成功添加的另一种方法是修改 *rango/views.py* 文件中的 `add_category()` 函数，把 `form.save(commit=True)` 改成 `cat = form.save(commit=True)`，为通过表单创建的分类对象提供一个引用，这样便可以在控制台中打印分类，例如 `print(cat, cat.slug)`。

清理表单数据

你或许还记得，`Page` 模型有个 `url` 属性，其类型为 `URLField`。在对应的 HTML 表单中，Django 期望这个字段中的文本是格式正确的完整 URL。然而，用户可能觉得输入 *http://www.url.com* 这样的 URL 很麻烦，甚至根本不知道何为正确的 URL 格式。

■ URL 检查 ■

多数现代浏览器会检查 URL 的格式是否正确，因此这里所说的情况只针对旧浏览器。虽然如此，但是其中涉及如何在存入数据库之前清理数据。如果你没有旧浏览器做试验（或者你不相信），可以把 `URLField` 更换成 `CharField`，这样渲染得到的 HTML 便不会指示浏览器做检查，我们编写的清理代码就能执行。

为了防止用户输入的数据有错，我们可以在 `ModelForm` 子类中覆盖 `clean()` 方法。这个方法在数据存为新模型实例之前调用，因此在这一时刻特别适合确认（甚至修正）用户在表单中输入的数据。我们可以检查用户在 `url` 字段中输入的值是否以 `http://` 开头，如果不是，在用户输入的值前面加上 `http://`。

```
class PageForm(forms.ModelForm):
    ...
    def clean(self):
        cleaned_data = self.cleaned_data
        url = cleaned_data.get('url')

        # 如果 url 字段不为空，而且不以“http://”开头
        # 在前面加上“http://”
        if url and not url.startswith('http://'):
            url = 'http://' + url
            cleaned_data['url'] = url

            return cleaned_data
```

在 clean() 方法中，基本的处理过程是一样的，你可以轻易套用，按照自己的方式处理表单数据。

❶ 表单数据从 cleaned_data 字典中获取。

❷ 要检查的表单字段使用字典的 .get() 方法从 cleaned_data 字典中获取。如果用户未在某个表单字段中输入值，那么 cleaned_data 字典中便没有对应的键值对。此时，.get() 方法返回 None，而不抛出 KeyError 异常。这样能让代码稍微简洁一点。

❸ 先检查要处理的表单字段中有没有值，如果有再检查值是什么。如果发现值不符合预期，那就做些处理，然后把新值存入 cleaned_data 字典。

❹ clean() 方法的最后必须返回 cleaned_data 字典的引用。否则改动不生效。

这个简单的示例展示了如何在保存之前清理数据。clean() 方法特别有用，尤其适合为字段提供默认值，或者为用户没有填写的字段提供值。

★ 覆盖方法 ★

覆盖 Django 框架的方法能为你的应用添加额外的功能。除了 ModelForm 类的 clean() 方法之外，还有很多方法可以放心覆盖，详情参见 Django 文档。

</> 练习

本章内容结束了，请看下面的问题，想一想应该怎么解决。

- 在添加分类表单中如果不输入分类名称会发生什么？
- 如果添加的分类已经存在会发生什么？
- 如果访问不存在的分类会发生什么？后文有提示。
- 定义 ModelForm 子类时，我们为某些字段设定了 max_length 参数，这与模型中的验证重复了。想想如何重构，以免重复指定 max_length 值？
- 阅读 Django 官方教程第四部分，巩固本章所学。
- 让用户能在分类中添加网页。参见下面的示例代码和提示。

创建添加网页的视图、模板和 URL 映射

接下来应该让用户能在分类中添加网页。基本步骤与添加分类一样。

- 编写一个视图，`add_page()`
- 创建一个模板，*rango/add_page.html*
- 添加一个 URL 映射
- 更新分类页面的模板和视图，加上添加网页链接

下面给出 `add_page()` 视图函数，给你一点提示。

```
from rango.forms import PageForm

def add_page(request, category_name_slug):
    try:
        category = Category.objects.get(slug=category_name_slug)
    except Category.DoesNotExist:
        category = None

    form = PageForm()
    if request.method == 'POST':
        form = PageForm(request.POST)
        if form.is_valid():
            if category:
                page = form.save(commit=False)
                page.category = category
                page.views = 0
                page.save()
                return show_category(request, category_name_slug)
        else:
            print(form.errors)

    context_dict = {'form':form, 'category': category}
    return render(request, 'rango/add_page.html', context_dict)
```

★ 提示 ★

如果你觉得练习有难度，下面给你一些提示。

- 在 *add_page.html* 模板中可以通过 `{{ category.slug }}` 访问分类的别名，因为视图通过上下文字典把 `category` 对象传给模板了。
- 仅在请求的分类存在时才显示链接。在模板中使用 `if` 标签判断：`{% if cat %} .... {% else %} A category by this name does not exist {% endif %}`。
- 在 Rango 的 *category.html* 模板中添加一个链接，后面带有换行：`<a href="/rango/category/{{category.slug}}/add_page/">Add Page</a> <br/>`。
- *add_page.html* 模板中的表单要把请求发给 */rango/category/{{ category.slug }}/add_page/*。
- 更新 *rango/urls.py*，添加一个 URL 映射处理上述链接。
- 为了避免重复设定 `max_length` 参数，可以在 `Category` 类中声明一个属性，用它设定最大长度限制，然后在需要的地方引用这个属性。

如果实在不知道怎么做，请查看 GitHub 中的代码。

8 模板进阶

目前，我们为 Rango 应用的几个页面创建了 HTML 模板。你可能发现了，不同模板之间有很多 HTML 代码是重复的，这违背了 DRY 原则。此外，你可能也注意到了，链接中使用的是硬编码的 URL 路径。这些问题会导致网站难以维护，倘若想改变网站的整体结构，或者调整 URL 路径，每个模板都要修改。

本章使用模板继承解决第一个问题，使用 URL 模板标签解决第二个问题。先看第二个问题。

8.1 使用相对 URL

目前，模板中的链接地址使用的是硬编码的 URL，例如 `<a href="/rango/about/">About</a>`。这样做的缺点是一旦修改了 *urls.py* 中的 URL 映射，就要更新对应的所有 URL 引用。正确的方法是使用模板标签 `url` 查询 *urls.py* 文件中的 URL 模式，动态插入 URL 路径。

在模板中使用相对 URL 十分简单。若想链接到关于页面，可以在模板中插入下面这行代码：

```
<a href="{% url 'about' %}">About</a>
```

Django 模板引擎会检查 *urls.py* 模块中有没有 `name` 属性的值为 `about` 的 URL 模式，然后反向匹配 URL。这样一来，如果修改了 *urls.py* 模块中的 URL 映射，无需一个一个修改模板。

URL 模式也可以不通过名称引用，而是直接引用视图，如下所示：

```
<a href="{% url 'rango.views.about' %}">About</a>
```

此时，必须保证 `rango` 应用的 *views.py* 模块中有名为 `about` 的视图。

Rango 应用的 *index.html* 模板中有个带参数的 URL，即 */rango/category/{{ category.slug }}*。为了避免硬编码，我们可以使用 `url` 模板标签，并指定 URL（或视图）的名称和分类的别名，如下所示：

```
{% for category in categories %}
    <li>
        <a href="{% url 'show_category' category.slug %}">
            {{ category.name }}
        </a>
    </li>
{% endfor %}
```

别忙着把所有模板中硬编码的 URL 都更改为相对 URL，在此之前我们先利用模板继承调整模板的结构，去除重复。

■ URL 和多个 Django 应用 ■

本书只专注于开发一个 Django 应用，即 Rango。然而，你的 Django 项目中可能有多个应用，说不定有几百个 URL。这就引出一个问题：这么多 URL 要怎么组织呢？毕竟两个应用中可能存在同名视图，从而导致冲突。

为了解决这个问题，Django 为各应用的 URL 配置模块提供了命名空间。在应用的 *urls.py* 模块中加上 `app_name` 变量就设定了命名空间。下述示例把 Rango 应用的命名空间设为 `rango`。

```
from django.conf.urls import url
from rango import views

app_name = 'rango'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    ...
]
```

添加 `app_name` 变量后，要像下面这样引用 Rango 应用中的 URL：

```
<a href="{% url 'rango:about' %}">About</a>
```

`url` 标签中的冒号负责分开命名空间和 URL 名称。

这是个高级功能，同一个项目中有多个应用时才能用得到，不过现在知道也没什么坏处。

8.2 去除重复

几乎每个专业的网站都有一系列重复的组件，例如页头、侧边栏和页脚，但是每个页面都重复编写这些组件的 HTML 显然是不明智的。试想，如果要调整网站的页头呢？你要修改每个页面，换用新的页头。这是个费时的工作，而且可能出现人为错误。

为免浪费时间复制粘贴 HTML 标记，我们可以利用 Django 模板引擎提供的模板继承功能尽量避免重复。

使用模板继承的基本步骤如下。

1. 找出模板中重复出现的部分，例如页头、侧边栏、页脚和内容区。有时，你可以把各页面的结构画在纸上，这样便于找出通用的部分。
2. 创建一个*基模板*（base template），实现页面的基本骨架结构，提供通用的部分（例如页头的徽标和标题，页脚的版权声明），并定义一些*区块*（block），以便在不同的页面调整所显示的内容。
3. 为应用的不同页面创建专门的模板，都继承自基模板，然后指定各区块的内容。

在基模板中定义重复出现的 HTML

显然，目前我们创建的模板有很多重复的 HTML 代码。下面把各页面显示的具体内容剔除，得到各模板重复使用的骨架结构。

```
<!DOCTYPE html>
{% load staticfiles %}

<html>
    <head lang="en">
        <meta charset="UTF-8" />
        <title>Rango</title>
    </head>
    <body>
        <!-- 各页面的具体内容 -->
    </body>
</html>
```

我们暂且把这个 HTML 页面作为 Rango 应用的基模板。把上述代码保存在 *templates/rango/* 目录中的 *base.html* 文件里。

◆ DOCTYPE 放在开头 ◆

记住，<!DOCTYPE html> 声明必须放在模板的第一行。如若不然，渲染得到的页面可能不符合 W3C HTML 指导方针。

定义区块

创建好基模板之后，接下来要指明模板中的哪些部分可由继承它的模板覆盖。为此，要使用 block 标签。例如，可以像下面这样在 *base.html* 模板中添加 body_block 区块：

```
<!DOCTYPE html>
{% load staticfiles %}

<html>
    <head lang="en">
        <meta charset="UTF-8" />
        <title>Rango</title>
    </head>
    <body>
        {% block body_block %}
        {% endblock %}
    </body>
</html>
```

还记得吗，Django 模板标签放在 {% 和 %} 之间。因此，区块以 {% block <NAME> %} 开头，其中 <NAME> 是区块的名称。区块必须以 endblock 结尾，而且也要放在 {% 和 %} 之间，即 {% endblock %}。

可以为区块指定默认内容，在子模板没有提供该区块的内容时使用（见 8.3 节）。指定默认内容的方法是在 {% block %} 和 {% endblock %} 之间添加 HTML 标记，如下所示。

```
{% block body_block %}
    This is body_block's default content.
{% endblock %}
```

创建各页面的模板时，我们将继承 *rango/base.html* 模板，然后覆盖 body_block 区块的内容。模板中的区块数量不限，可以根据需要定义。例如，可以创建页面标题区块、页脚区块、侧边栏区块，等等。区块是 Django 模板系统一个特别强大的功能，详情参见 Django 文档。

★ 提取通用结构 ★

在基模板中要尽量提取重复出现的内容。这样做看起来麻烦，但是后期节省的维护时间远比这多。

思考是痛苦的，但总比做很多单调的工作要好！

进一步抽象

了解区块的作用之后，下面进一步抽象基模板。打开 *rango/base.html* 模板，像下面这样修改：

```
<!DOCTYPE html>
{% load staticfiles %}

<html>
    <head>
        <title>
            Rango -
            {% block title_block %}
                How to Tango with Django!
            {% endblock %}
        </title>
    </head>
    <body>
        <div>
            {% block body_block %}
            {% endblock %}
        </div>
        <hr />
        <div>
            <ul>
                <li><a href="{% url 'add_category' %}">Add New Category</a></li>
                <li><a href="{% url 'about' %}">About</a></li>
                <li><a href="{% url 'index' %}">Index</a></li>
            </ul>
        </div>
    </body>
</html>
```

上述代码对基模板做了两处改动：

- 首先是增加了 title_block 区块。这样在子模板中可以为各页面定制标题。如果子模板没有覆盖这个区块，使用默认值 How to Tango with Django!，得到的页面标题为 Rango - How to Tango with Django!。请仔细看一下上述代码片段中的 <title> 标签。
- 把 *index.html* 模板中的几个链接放在 HTML <div> 标签里，放到 body_block 区块下面。这样所有继承的子模板都将具有这些链接。这组链接前面有条水平线（<hr />），目的是从视觉上把链接与 body_block 区块分开。

8.3 模板继承

创建好基模板之后，接下来要更新其他模板，让它们继承基模板。先重构 *rango/category.html* 模板。

首先把模板中重复的 HTML 代码删除，只留下对所在页面有用的模板标签或命令。然后在模板的开头添加下面这行代码：

```
{% extends 'rango/base.html' %}
```

extends 命令接受一个参数，即要扩展（继承）的模板。这个参数的值是个路径，相对项目的 *templates* 目录。例如，Rango 应用中的模板都继承 *rango/base.html*，而不是 *base.html*。继续修改 *category.html* 模板，如下所示：

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    {{ category.name }}
{% endblock %}

{% block body_block %}
    {% if category %}
        <h1>{{ category.name }}</h1>

        {% if pages %}
            <ul>
            {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
```

```
            {% endfor %}
            </ul>
        {% else %}
            <strong>No pages currently in category.</strong>
        {% endif %}
        <a href="{% url 'add_page' category.slug %}">Add a Page</a>
    {% else %}
        The specified category does not exist!
    {% endif %}
{% endblock %}
```

◆ 加载 staticfiles ◆

用到静态文件的每个模板都要在文件顶部添加 `{% load staticfiles %}`。否则会导致错误！对 Django 模板而言，必须在用到模块的每个模板中导入模块。这与面向对象编程语言（例如 Java）有所不同。在面向对象编程语言中，导入的模块会沿着类继承体系向下延伸。注意我们是如何使用 `url` 模板标签引用 *rango/<category-name>/add_page/* URL 模式的。`category.slug` 作为参数传给 `url` 模板标签，Django 模板引擎会据此生成正确的 URL。

现在 *category.html* 模板继承自 *rango/base.html*，并扩展了 `title_block` 和 `body_block` 区块，变得简洁多了。*category.html* 模板不必是完整的 HTML 文档，因为 *base.html* 提供了所需的结构。我们只需在基模板的基础上添加所需的内容，这样便能渲染得到完整的 HTML 文档，发给客户端。渲染得到的 HTML 文档符合标准，包含所需的各部分标记，例如第一行的文档类型声明。

★ 深入了解模板 ★

这里我们只说明了如何尽量减少模板中重复的 HTML 结构。然而，Django 模板语言还有很多强大的功能，甚至可以自定义模板标签。

利用模板还能减少应用视图的代码。例如，如果你想在每个页面中展示从数据库中获取的特定内容，可以专门定义一个视图，在对应的模板中展示这部分内容。这样就无需在需要显示这部分内容的每个模板中都调用 Django ORM 函数获取所需的数据。

现在，请花点时间阅读 Django 文档中有关模板的说明。

8.4 render() 函数和 request 上下文

视图有多种渲染模板的方式，不过首选的是 Django 提供的简洁方式，即 render() 函数。render() 函数的第一个参数是 request 上下文，里面有大量信息，包括会话和用户等等，详情参见 Django 文档。把 request 传给模板意味着，我们可以在模板中访问其中的信息。下一章将访问关于用户的信息。现在请检查一下所有视图，确保都是用 render() 函数渲染模板的。如若不然，后面就得不到所需的信息。

★ 如何检查？ ★

下面以 about() 视图为例说明如何检查和修改。about() 视图目前的实现方式是硬编码响应字符串，如下所示。注意，我们只是发送字符串，没有用到传入视图的 request 参数。

```
def about(request):
    return HttpResponse('Rango says: Here is the about page.
                         <a href="/rango/">Index</a>')
```

为了改用模板，我们要调用 render() 函数，并传入 request 对象。这样修改之后，模板引擎就能访问关于请求（例如请求类型，GET 或 POST）和用户（例如用户的状态，参见第 9 章）的信息了。

```
def about(request):
    # 打印请求方法，是 GET 还是 POST
    print(request.method)
    # 打印用户名，如未登录，打印“AnonymousUser”
    print(request.user)
    return render(request, 'rango/about.html', {})
```

注意，render() 函数的最后一个参数是上下文字典，用于把额外的数据传给 Django 模板引擎。因为这里没什么额外数据要传给模板，所以使用一个空字典。

8.5 自定义模板标签

如果能在每个页面的侧边栏中展示有哪些分类可浏览就好了。根据目前所学，这个想法可以这样实现：

- 在 *base.html* 模板中添加一些代码，显示分类列表

- 在每个视图中通过 `Category` 对象获取所有分类，通过上下文字典传给模板

可是这样的实现方式非常不好，因为我们要在所有视图中重复添加相同的代码。为了不违背 DRY 原则，我们可以自定义模板标签，把相关的操作封装起来。

定义模板标签

新建 *rango/templatetags* 目录，然后在其中新建两个模块：一个命名为 *__init__.py*，内容为空；另一个命名为 *rango_template_tags.py*，写入下述代码。

```
from django import template
from rango.models import Category

register = template.Library()

@register.inclusion_tag('rango/cats.html')
def get_category_list():
    return {'cats': Category.objects.all()}
```

这段代码定义了一个名为 `get_category_list()` 的函数，返回结果为分类列表。但是从 `register.inclusion_tag()` 装饰器可以看出，这个函数需要 *rango/cats.html* 模板的支持。创建这个模板，写入下述 HTML 标记。

```
<ul>
{% if cats %}
    {% for c in cats %}
        <li><a href="{% url 'show_category'  c.slug %}">{{ c.name }}</a></li>
    {% endfor %}
{% else %}
    <li><strong>There are no categories present.</strong></li>
{% endif %}
</ul>
```

为了在 *base.html* 模板中使用这个模板标签，首先要在文件顶部添加 `{% load rango_template_tags %}`。然后创建一个区块，表示侧边栏。在侧边栏中通过下述代码调用我们自定义的模板标签。

```
<div>
    {% block sidebar_block %}
        {% get_category_list %}
```

```
    {% endblock %}
</div>
```

试试看。现在继承自 *base.html* 模板的每个页面都将显示分类列表（稍后移到侧边）。

■ 重启服务器 ■

修改模板标签后要重启 Django 开发服务器，否则 Django 不会注册新标签。

参数化模板标签

为了提高灵活性，可以参数化模板标签。举个例子：突出显示当前查看的分类。添加参数很简单，参照下述代码修改 `get_category_list()` 函数。

```
def get_category_list(cat=None):
    return {'cats': Category.objects.all(),
            'act_cat': cat}
```

注意，`cat` 参数是可选的，如果未传入，默认为 `None`。

接下来修改 *base.html* 模板，传入当前查看的分类（仅当存在时）。

```
<div>
    {% block sidebar_block %}
        {% get_category_list category %}
    {% endblock %}
</div>
```

此外，还要修改 *cats.html*。

```
{% for c in cats %}
    {% if c == act_cat %}
        <li>
        <strong>
              <a href="{% url 'show_category' c.slug %}">{{ c.name }}</a>
        </strong>
        </li>
    {% else  %}
        <li>
```

```
            <a href="{% url 'show_category' c.slug %}">{{ c.name }}</a>
        </li>
    {% endif %}
{% endfor %}
```

这里我们检查显示的分类与 `for` 循环当前遍历的分类是否相同（`c == act_cat`），如果相同，通过 `<strong>` 标签加粗，突出显示当前查看的分类名。

8.6 小结

本章讨论了下述内容：

- 使用 `url` 模板标签避免硬编码 URL
- 借助模板继承减少样板代码量
- 通过自定义模板标签减少视图中重复的代码

经过这些改进之后，模板代码比以前更简洁，也更易维护了。当然，Django 模板的功能还有很多，详情参阅 Django 文档。

</> 练习

读完本章之后请完成以下练习，巩固学到的 Django 模板知识。

- 修改 Rango 应用中的其他模板，继承 *base.html* 模板。具体过程参见前文。改完之后，所有模板都继承自 *base.html*。
- 修改的过程中顺便把 *index.html* 模板中的链接删除。那些链接用不到了！*about.html* 模板中指向 Rango 首页的链接可以一并删除。
- 修改 *index.html* 模板时，不要删除图像。
- 使用 `url` 模板标签更新所有内部链接。也可以在 *views.py* 模块中修改，此时要使用 `reverse()` 辅助函数。

★ 提示 ★

- 先重构 *about.html* 模板。
- 更新每个模板中的 `title_block` 和 `body_block` 区块。
- 在修改的过程中始终运行着开发服务器，随时检查改动后的页面。不要改完之后再检查。边改边测试更安全。
- 若想指向某个分类的页面，可以使用下述模板代码。请留意其中的 `{% url %}` 命令。

```
<a href="{% url 'show_category' category.slug %}">{{ category.name }}</a>
```

9 用户身份验证

接下来的三章为您介绍 Django 提供的用户身份验证机制。我们将使用 `django.contrib.auth` 包中的 `auth` 应用。根据 Django 文档，这个应用提供了下述概念和功能：

- 用户和用户模型
- 权限，判断用户可以做什么及不可以做什么的旗标（是/否）
- 用户组，把相关权限一次赋予多个用户
- 可配置的密码哈希系统，保证数据安全不可或缺
- 登录或限制性内容所需的表单和视图

在用户身份验证方面，Django 提供了足够的机制。本章介绍基础知识，让你了解可用的工具和底层概念。

9.1 设置身份验证

在使用 Django 提供的身份验证机制之前，要在项目的 *settings.py* 文件中添加相关的设置。

在 *settings.py* 文件中找到 `INSTALLED_APPS` 列表，检查有没有列出 `django.contrib.auth` 和 `django.contrib.contenttypes`。`INSTALLED_APPS` 列表应该类似下面这样：

```
INSTALLED_APPS =[
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
        'rango',
    ]
```

django.contrib.auth 用于访问 Django 提供的身份验证系统，而 django.contrib.contenttypes 供 auth 应用跟踪数据库中的模型。

★ 如有必要，迁移 ★

如果 INSTALLED_APPS 列表中没有 django.contrib.auth 和 django.contrib.contenttypes，要自己添加，添加后还要执行 python manage.py migrate 命令更新数据库，添加所需的表，例如 User 模型的表。

在 Django 项目中添加新的应用后，一般最好执行 migrate 命令，以防应用中有模型要与底层数据库同步。

9.2 密码哈希

任何情况下都不能在数据库中存储明文密码。[1] 如果包含用户账户的数据库落到不怀好意的人手上，可能造成天大的灾难。幸好，Django 的 auth 应用默认存储的是经过 PBKDF2 算法计算过的密码哈希值，可以有效保护用户数据的安全。然而，如果你想进一步控制生成密码哈希值的方式，可以在项目的 *settings.py* 模块中更换 Django 使用的算法。为此，添加 PASSWORD_HASHERS 元组，例如：

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
)
```

罗列哈希算法的顺序很重要，Django 将使用 PASSWORD_HASHERS 中的第一个哈希算法（settings.PASSWORD_HASHERS[0]）。如果第一个无效，而且还有其他哈希算法供选择，Django 将依次尝试后面的算法。

如果想使用更为安全的哈希算法，可以安装 Bcrypt（pip install bcrypt），然后把 PASSWORD_HASHERS 设为：

1. https://stackoverflow.com/q/1197417

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
]
```

前面说过，Django 默认使用 PBKDF2 算法计算密码哈希值。如果未在 *settings.py* 文件中设定 `PASSWORD_HASHERS`，Django 将使用默认的 `PBKDF2PasswordHasher`。详情参见 Django 文档。

9.3 密码验证器

鉴于人们经常使用相对容易猜出的密码，Django 1.9 引入了一个备受期待的新功能——密码验证。在 Django 项目的 *settings.py* 模块中有个字典构成的列表，名为 `AUTH_PASSWORD_VALIDATORS`。在嵌套的字典中可以清楚地看出，Django 1.9 自带了一些常用的密码验证器，例如针对长度的验证器。每个验证器都有 `OPTIONS` 字典，以便自定义选项。假如你想确保密码最短为 6 个字符，那么可以把 `MinimumLengthValidator` 的 `min_length` 选项设为 `6`，如下所示：

```
AUTH_PASSWORD_VALIDATORS = [
    ...
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': { 'min_length': 6, }
    },
    ...
]
```

除了自带的密码验证器之外，还可以自定义。本书不介绍具体方法，如果感兴趣，请阅读 Django 文档。

9.4 User 模型

`User` 对象（`django.contrib.auth.models.User`）是 Django 身份验证系统的核心，表示与 Django 应用交互的每个个体。根据 Django 文档，身份验证系统的很多方面都能用到 `User` 对象，例如访问限制、注册新用户，以及网站内容与创建者之间的关系。

`User` 模型有 5 个主要属性：

- 用户账户的用户名（username）
- 用户账户的密码（password）
- 用户的电子邮件地址（email）
- 用户的名字（first_name）
- 用户的姓（last_name）

此外，User 模型还有其他属性，例如 is_active、is_staff 和 is_superuser。这些属性的值都是布尔值，分别用于指明账户是否激活、是否为团队成员，以及是否拥有超级用户权限。User 模型的完整属性列表参阅 Django 文档。

9.5 增加用户属性

除了 User 模型提供的属性之外，如果还需要其他用户相关的属性，要自己定义一个与 User 模型关联的模型。对 Rango 应用而言，我们想为用户账户增加两个属性：

- 一个 URLField，让 Rango 的用户设定自己的网站
- 一个 ImageField，让 Rango 的用户设定自己的头像

为此，要在 Rango 应用的 *models.py* 文件中定义一个模型。我们把这个模型命名为 UserProfile。

```
class UserProfile(models.Model):
    # 这一行是必须的
    # 建立与 User 模型之间的关系
    user = models.OneToOneField(User)

    # 想增加的属性
    website = models.URLField(blank=True)
    picture = models.ImageField(upload_to='profile_images', blank=True)

    # 覆盖 __str__() 方法，返回有意义的字符串
    # 如果使用 Python 2.7.x，还要定义 __unicode__ 方法
    def __str__(self):
        return self.user.username
```

注意，这个模型与 User 模型之间建立的一对一关系。因为引用了默认的 User 模型，所以要在 *models.py* 文件中导入它：

```
from django.contrib.auth.models import User
```

我们为 Rango 应用的用户账户增加了两个字段，还提供了 __str__() 方法，以便在需要 UserProfile 实例的字符串表示形式时返回有意义的值。注意，使用 Python 2 的话，还要定义 __unicode__() 方法，返回用户名的 Unicode 格式。

我们增加的 website 和 picture 字段都设定了 blank=True。因此这两个字段都可以为空，不是必须要提供值。

此外，注意 ImageField 字段的 upload_to 参数。这个参数的值与项目的 MEDIA_ROOT 设置（第 4 章）结合在一起，确定上传的头像存储在哪里。假如 MEDIA_ROOT 的值为 <workspace>/tango_with_django_project/media/，upload_to 参数的值为 profile_images，那么头像将存储在 *<workspace>/tango_with_django_project/media/profile_images/* 目录中。

■ 能通过继承扩展吗？ ■

你可能想通过继承 User 模型增加字段，但是考虑到其他应用或许也会访问 User 模型，因此不建议使用继承，而是推荐在数据库中建立一对一关系。

★ 安装 PIL ★

Django 的 ImageField 字段要使用 Python Imaging Library（PIL）。如果你还未安装，执行 pip install pillow 命令，通过 pip 安装 PIL。如果未启用 JPEG 支持，也可以安装 PIL：pip install pillow --global-option="build_ext" --global-option="--disable-jpeg"。

若想查看（虚拟）环境中安装了哪些包，执行 pip list 命令。

如果想通过 Django 管理界面访问 UserProfile 模型数据，在 Rango 应用的 *admin.py* 模块中导入 UserProfile 模型：

```
from rango.models import UserProfile
```

然后注册模型：

```
admin.site.register(UserProfile)
```

★ 别忘了迁移 ★

定义新模型后必须更新数据库，别忘了！在终端或命令提示符中执行 `python manage.py makemigrations rango` 命令，为新增的 `UserProfile` 模型创建迁移脚本。然后执行 `python manage.py migrate` 命令，运行迁移，在底层数据库中创建相关的表。

9.6 创建用户注册视图和模板

一切准备妥当之后，接下来实现用户注册功能。为此，我们要定义一个新视图、创建一个模板，并添加一个 URL 映射。

■ 现成的用户注册应用 ■

注意，有一些现成的用户注册应用可以拿来直接使用，无需我们自己动手实现注册和登录功能。

然而，在使用这样的应用之前最好了解一下底层机制。没有痛苦就没有收获。在这个过程中还能巩固你对表单的理解，学会如何扩展 `User` 模型，以及如何上传媒体文件。

实现用户注册功能的步骤如下：

- 定义 `UserForm` 和 `UserProfileForm`
- 添加一个视图，处理创建新用户的过程
- 创建一个模板，显示 `UserForm` 和 `UserProfileForm`
- 把 URL 映射到前面添加的视图上

最后，还要在首页添加一个链接，指向注册页面。

定义 UserForm 和 UserProfileForm

我们要在 *rango/forms.py* 中定义两个类，继承自 `forms.ModelForm`。其中一个针对 `User` 类，另一个针对前面定义的 `UserProfile` 模型。这两个 `ModelForm` 子类创建的 HTML 表单用于显示相应模型的字段，为我们节省了很多工作量。

首先，在 *rango/forms.py* 文件中定义两个继承自 `forms.ModelForm` 的类。

```python
class UserForm(forms.ModelForm):
    password = forms.CharField(widget=forms.PasswordInput())

    class Meta:
        model = User
        fields = ('username', 'email', 'password')

class UserProfileForm(forms.ModelForm):
    class Meta:
        model = UserProfile
        fields = ('website', 'picture')
```

注意，这两个类中都有 `Meta` 类。如其名所示，`Meta` 类的作用是为所在的类提供额外的属性。`Meta` 类中必须有 `model` 字段。`UserForm` 类对应的模型是 `User`。此外，还要通过 `fields` 或 `exclude` 指定要在表单中显示或排除的字段。

这里，我们只想显示 `User` 模型的 `username`、`email` 和 `password` 字段，以及 `UserProfile` 模型的 `website` 和 `picture` 字段。`UserProfile` 模型的 `user` 字段在注册用户时设定。这是因为创建 `UserProfile` 实例时，还没有 `User` 实例可用。

此外，注意 `UserForm` 中定义了 `password` 属性。虽然 `User` 模型实例有 `password` 属性，但是在渲染的 HTML 表单中这个字段的值不会被遮盖，用户输入的密码是可见的。鉴于此，我们重新定义了 `password` 属性，指定使用 `PasswordInput()` 小组件显示这个 `CharField`，以防用户输入的密码被人窥见。

最后，别忘了在 *forms.py* 模块的顶部导入所需的类。为了便于你参考，下面给出导入语句：

```python
from django import forms
from django.contrib.auth.models import User
from rango.models import Category, Page, UserProfile
```

定义 `register()` 视图

接下来要渲染表单及处理表单数据。在 Rango 应用的 *views.py* 模块中，添加一个 `import` 语句，导入新定义的 `UserForm` 和 `UserProfileForm` 类。

```python
from rango.forms import UserForm, UserProfileForm
```

然后定义 `register()` 视图：

```python
def register(request):
    # 一个布尔值，告诉模板注册是否成功
    # 一开始设为 False，注册成功后改为 True
    registered = False

    # 如果是 HTTP POST 请求，处理表单数据
    if request.method == 'POST':
        # 尝试获取原始表单数据
        # 注意，UserForm 和 UserProfileForm 中的数据都需要
        user_form = UserForm(data=request.POST)
        profile_form = UserProfileForm(data=request.POST)

        # 如果两个表单中的数据是有效的……
        if user_form.is_valid() and profile_form.is_valid():
            # 把 UserForm 中的数据存入数据库
            user = user_form.save()

            # 使用 set_password 方法计算密码哈希值
            # 然后更新 user 对象
            user.set_password(user.password)
            user.save()

            # 现在处理 UserProfile 实例
            # 因为要自行处理 user 属性，所以设定 commit=False
            # 延迟保存模型，以防出现完整性问题
            profile = profile_form.save(commit=False)
            profile.user = user

            # 用户提供头像了吗？
            # 如果提供了，从表单数据库中提取出来，赋给 UserProfile 模型
            if 'picture' in request.FILES:
                profile.picture = request.FILES['picture']

            # 保存 UserProfile 模型实例
            profile.save()

            # 更新变量的值，告诉模板成功注册了
            registered = True
        else:
            # 表单数据无效，出错了？
```

```
                # 在终端打印问题
                print(user_form.errors, profile_form.errors)
    else:
        # 不是 HTTP POST 请求，渲染两个 ModelForm 实例
        # 表单为空，待用户填写
        user_form = UserForm()
        profile_form = UserProfileForm()

    # 根据上下文渲染模板
    return render(request,
                  'rango/register.html',
                  {'user_form': user_form,
                   'profile_form': profile_form,
                   'registered': registered})
```

这个视图看似复杂，其实与添加分类和添加网页的视图十分相似。这里，我们要处理两个不同的 ModelForm 实例，一个针对 User 模型，一个针对 UserProfile 模型。如果用户上传了头像，还要处理头像图片。

此外，还要建立两个模型实例之间的关系。为此，我们先创建 User 模型实例，然后在 UserProfile 实例中引用它：profile.user = user。UserProfileForm 表单的 user 属性必须这样设定，不能让用户自行填写。

创建注册页面的模板

现在要创建 register() 视图的模板。新建 *rango/register.html* 文件，写入下述代码。

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    Register
{% endblock %}

{% block body_block %}
    <h1>Register for Rango</h1>
    {% if registered %}
        Rango says: <strong>thank you for registering!</strong>
        <a href="{% url 'index' %}">Return to the homepage.</a><br />
```

```
    {% else %}
        Rango says: <strong>register here!</strong><br />
        <form id="user_form" method="post" action="{% url 'register' %}"
              enctype="multipart/form-data">

        {% csrf_token %}

        <!-- 显示每个表单 -->
        {{ user_form.as_p }}
        {{ profile_form.as_p }}

        <!-- 提供一个按钮，点击后提交表单 -->
        <input type="submit" name="submit" value="Register" />
    </form>
    {% endif %}
{% endblock %}
```

★ 使用 url 模板标签 ★

注意，上述模板中使用了 url 模板标签，例如 {% url 'register' %}。因此，后面添加的 URL 映射要命名为 register。

注意，这个模板使用视图中的 registered 变量判断注册是否成功。registered 的值为 False 时，显示注册表单；否则，显示成功注册消息。

此外，我们在 user_form 和 profile_form 上调用了 as_p 模板函数。这么做的目的是在段落（HTML <p> 标签）中显示各个表单元素，一行显示一个表单元素。

最后注意，我们为 <form> 元素设定了 enctype 属性。这是因为，如果用户想上传头像，表单数据中将包含二进制数据，而且可能相当大。传给服务器时，这些数据要分成几部分。因此，我们要设定 enctype="multipart/form-data"，让 HTTP 客户端（Web 浏览器）分段打包和发送数据。如若不然，服务器收不到用户提交的全部数据。

◆ 分段报文和二进制文件 ◆

注意 <form> 元素的 enctype 属性。如果想让用户通过表单上传文件，必须把 enctype 属性设

为 multipart/form-data。这样设定后，浏览器会以特殊的方式发送表单数据。说白了，就是把表示文件的数据分成多个小块，然后发送。详情参见 Stack Overflow 网站中这个优秀的回答。

此外，别忘了在表单中添加 CSRF 令牌，即 {% csrf_token %}。否则，Django 的跨站请求伪造保护中间件将拒绝接收表单的内容，返回错误。

添加 URL 映射

有了视图和对应的模板之后，现在可以添加 URL 映射了。打开 Rango 应用的 *urls.py* 模块，根据下述代码修改 urlpatterns。

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'about/$', views.about, name='about'),
    url(r'^add_category/$', views.add_category, name='add_category'),

    url(r'^category/(?P<category_name_slug>[\w\-]+)/$',
        views.show_category,
        name='show_category'),

    url(r'^category/(?P<category_name_slug>[\w\-]+)/add_page/$',
        views.add_page,
        name='add_page'),

    url(r'^register/$',
        views.register,
        name='register'), # 新增的模式
]
```

新增的模式把 */rango/register/* URL 映射到 register() 视图上。注意，我们为这个新模式设定了 name 参数，以便在模板中使用 url 引用，例如 {% url 'register' %}。

添加链接

最后，在 *base.html* 模板中添加一个链接，指向注册页面。参照下述代码更新无序列表中的链接，在 Rango 应用的每个页面中都添加指向注册页面的链接。

```
<ul>
    <li><a href="{% url 'add_category' %}">Add a New Category</a></li>
    <li><a href="{% url 'about' %}">About</a></li>
    <li><a href="{% url 'index' %}">Index</a></li>
    <li><a href="{% url 'register' %}">Register Here</a></li>
</ul>
```

检验结果

一切就绪之后，该检验结果了。启动 Django 开发服务器，注册一个用户试试。如果愿意，上传一个头像。注册表单如图 9-1 所示。

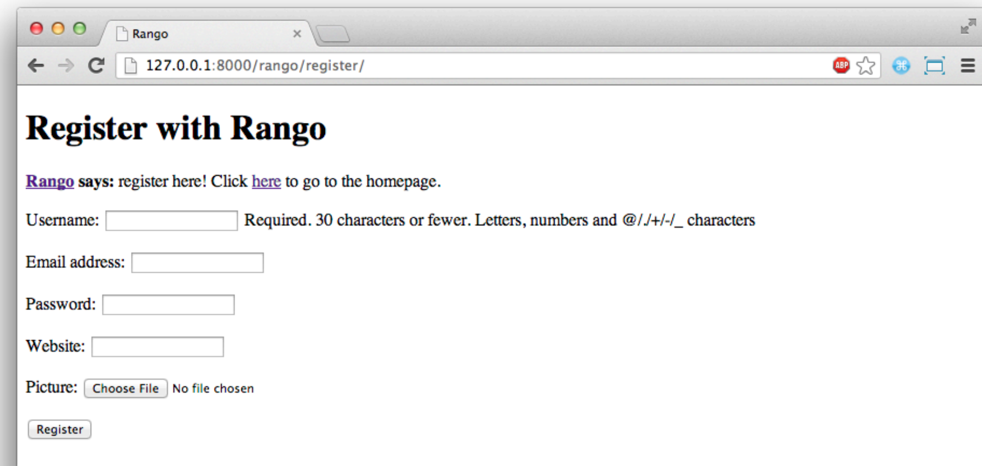


图 9-1：注册表单

看到成功注册消息后，User 和 UserProfile 模型在数据库中应该都会增加一条记录。在 Django 管理界面中确认是不是这样。

9.7 实现登录功能

用户能注册账户之后，接下来要让用户能够登录。为此，要执行以下几步：

- 定义一个视图，处理登录凭据
- 创建一个模板，显示登录表单

- 把登录视图映射到一个 URL 上
- 在首页添加登录链接

定义登录视图

首先，打开 Rango 应用的 *views.py* 模块，定义一个新视图，名为 `user_login()`。这个视图负责处理登录表单提交的数据，以及登入用户。

```
def user_login(request):
    # 如果是 HTTP POST 请求，尝试提取相关信息
    if request.method == 'POST':
        # 获取用户在登录表单中输入的用户名和密码
        # 我们使用的是 request.POST.get('<variable>')
        # 而不是 request.POST['<variable>']
        # 这是因为对应的值不存在时，前者返回 None，
        # 而后者抛出 KeyError 异常
        username = request.POST.get('username')
        password = request.POST.get('password')

        # 使用 Django 提供的函数检查 username/password 是否有效
        # 如果有效，返回一个 User 对象
        user = authenticate(username=username, password=password)

        # 如果得到了 User 对象，说明用户输入的凭据是对的
        # 如果是 None（Python 表示没有值的方式），说明没找到与凭据匹配的用户
        if user:
            # 账户激活了吗？可能被禁了
            if user.is_active:
                # 登入有效且已激活的账户
                # 然后重定向到首页
                login(request, user)
                return HttpResponseRedirect(reverse('index'))
            else:
                # 账户未激活，禁止登录
                return HttpResponse("Your Rango account is disabled.")
        else:
            # 提供的登录凭据有问题，不能登录
            print("Invalid login details: {0}, {1}".format(username, password))
            return HttpResponse("Invalid login details supplied.")
```

```
        # 不是 HTTP POST 请求，显示登录表单
        # 极有可能是 HTTP GET 请求
        else:
            # 没什么上下文变量要传给模板系统
            # 因此传入一个空字典
            return render(request, 'rango/login.html', {})
```

跟之前一样，因为要处理不同的情况，这个视图看起来十分复杂。从上述代码可以看出，`user_login()` 视图既能渲染登录表单（包含 `username` 和 `password` 两个字段），也能处理表单数据。

如果通过 HTTP GET 方法访问这个视图，显示登录表单。然而，如果通过 HTTP POST 请求访问，则处理表单数据。

如果通过 POST 请求发送有效的表单数据，从中提取用户名和密码。然后使用 Django 提供的 `authenticate()` 函数检查用户名和密码是否匹配某个用户账户。如果能找到这样的用户，返回一个 `User` 对象，否则返回 `None`。

返回 `User` 对象时，检查账户是否激活。如果是激活的，调用 Django 提供的 `login()` 函数，登入用户。

然而，如果发送的表达数据无效，例如用户名和密码没有都填，登录表单显示错误消息，提示用户名或密码无效。

你可能注意到了，这里用了一个新类，即 `HttpResponseRedirect`。从名称可以看出，`HttpResponseRedirect` 实例生成的响应让 Web 浏览器重定向到参数指定的 URL。注意，响应的 HTTP 状态码是表示重定向的 302，而不是表示成功的 200。详情参见 Django 文档。

最后，使用 Django 提供的 `reverse()` 函数获取 Rango 应用首页的 URL。`reverse()` 函数在 Rango 应用的 *urls.py* 模块中查找名为 `index` 的 URL 模式，解析出对应的 URL。如果以后修改了 URL 映射，视图代码不受影响。

`user_login()` 视图用到了 Django 提供的多个函数和类，因此要导入它们。下述 `import` 语句必须放在 *rango/views.py* 文件的顶部。

```
from django.contrib.auth import authenticate, login
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
```

创建登录页面的模板

有了视图之后，我们还要创建一个模板，让用户输入登录凭据。现在你应该知道要把模板放在 *templates/rango/* 目录中，不过这个模板的名称我不告诉你，请你根据 `user_login()` 视图的代码确定。在模板中写入下述代码：

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title_block %}
    Login
{% endblock %}

{% block body_block %}
<h1>Login to Rango</h1>
<form id="login_form" method="post" action="{% url 'login' %}">
    {% csrf_token %}
    Username: <input type="text" name="username" value="" size="50" />
    <br />
    Password: <input type="password" name="password" value="" size="50" />
    <br />
    <input type="submit" value="submit" />
</form>
{% endblock %}
```

`input` 元素的 `name` 属性要与 `user_login()` 视图中的保持一致。也就是说，用户名输入框的 `name` 属性的值应该是 `username`，而密码输入框的 `name` 属性的值应该是 `password`。此外，别忘了 `{% csrf_token %}`。

添加 URL 映射

创建好登录模板之后，接下来要把 `user_login()` 视图映射到一个 URL 上。修改 Rango 应用的 *urls.py* 模块，在 `urlpatterns` 列表中添加下述映射：

```
url(r'^login/$', views.user_login, name='login'),
```

添加链接

最后，添加一个链接，方便 Rango 的用户访问登录页面。编辑 *templates/rango/* 目录中的 *base.html*

模板，在无序列表中添加下述链接：

```
<ul>
    ...
    <li><a href="{% url 'login' %}">Login</a><li>
</ul>
```

如果愿意，还可以修改首页的页头，向已登录的用户显示独特的欢迎消息，而未登录的用户则显示一般的欢迎消息。在 *index.html* 模板中找到现在的欢迎消息：

```
hey there partner!
```

把这一行改成：

```
{% if user.is_authenticated %}
    howdy {{ user.username }}!
{% else %}
    hey there partner!
{% endif %}
```

可以看出，我们使用 `{% if user.is_authenticated %}` 检查用户是否通过身份验证。如果用户已登录，我们能访问 `user` 对象。因此，我们可以通过这个对象判断用户是否登录（验证身份）。如果用户已登录，我们便能获取关于用户的更多信息。在这个示例中，当用户登录后，我们获取用户的用户名。如果用户未登录，则显示一般的欢迎消息，即“hey there partner!”。

检验结果

启动 Django 开发服务器，尝试登录。登录前后首页的截图见图 9-2。

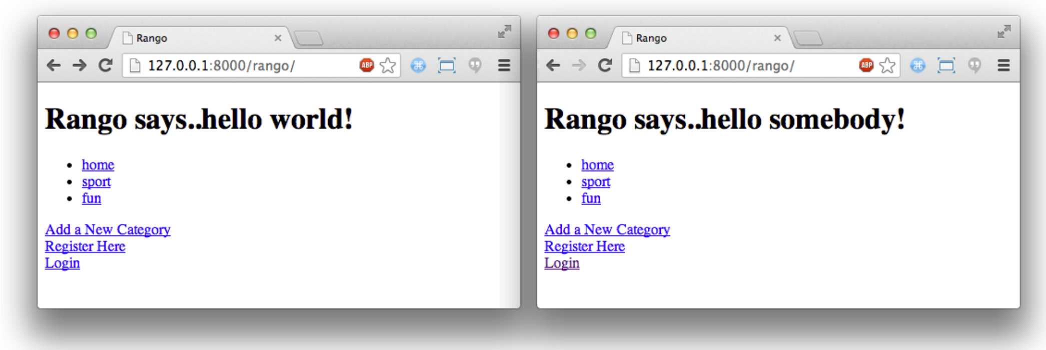


图 9-2：未登录的首页和登录后的首页（用户名为“somebody”）

至此，用户登录功能可用了。请启动 Django 开发服务器，注册一个新账户试试。成功注册后，你应该能使用自己设定的凭据登录。

9.8 限制访问

现在用户可以登录 Rango 应用了，根据客户的要求，接下来应该限制访问应用的某些页面，即只有注册的用户才能添加分类和网页。在 Django 中，这一要求有多种实现方式。

- 在模板中可以使用 `{% if user.authenticated %}` 模板标签修改页面渲染的内容
- 在视图中可以直接通过 `request` 对象检查用户有没有通过身份验证
- 此外，还可以使用 Django 提供的 `@login_required` 装饰器检查用户有没有通过身份验证

第二种方法使用 `user.is_authenticated()` 方法。`user` 对象通过传入视图的 `request` 对象访问。具体方法如下述代码所示：

```
def some_view(request):
    if not request.user.is_authenticated():
        return HttpResponse("You are logged in.")
    else:
        return HttpResponse("You are not logged in.")
```

第三种方法使用 Python 装饰器。装饰器根据同名软件设计模式命名，其作用是在不修改函数、方法或类源码的基础上调整函数、方法或类的功能。

Django 提供的 `login_required()` 装饰器可以依附在任何要求用户登录的视图上。如果未登录的用户尝试访问使用 `login_required()` 装饰的视图，浏览器会重定向到另一个页面；这个页面可以自己设定，通常是登录页面。

使用装饰器限制访问

举个例子。在 Rango 应用的 *views.py* 模块中添加一个视图，名为 `restricted()`：

```
@login_required
def restricted(request):
    return HttpResponse("Since you're logged in, you can see this text!")
```

注意，装饰器的用法是直接放在函数的签名上方，并在装饰器的名称前放一个 `@` 符号。Python 会先执行装饰器，再执行函数（方法）的代码。装饰器其实也是函数，因此如果装饰器在其他模块

中，要将其导入。因为 `login_required()` 装饰器经常使用，所以 *views.py* 模块的顶部经常有下述 `import` 语句。

```
from django.contrib.auth.decorators import login_required
```

然后在 Rango 应用的 *urls.py* 模块中添加一个 URL 映射：

```
url(r'^restricted/', views.restricted, name='restricted'),
```

此外，还要处理未登录的用户尝试访问 `restricted()` 视图的情况。此时应该怎么做呢？最简单的处理方法是重定向到他们能访问的页面，例如登录页面。这个页面的地址在项目的 *settings.py* 模块中设定。打开 *settings.py* 文件，定义 `LOGIN_URL` 变量，把值设为未登录时的重定向地址。这里，设为 */rango/login/* URL 上的登录页面：

```
LOGIN_URL = '/rango/login/'
```

这样设定后，`login_required()` 装饰器将把未登录的用户重定向到 */rango/login/* URL。

9.9 退出

用户能注册和登录之后，还要能退出。Django 提供的 `logout()` 函数能确保用户正确且安全地退出。用户的会话结束后，如果再想访问受限制的视图，要重新登录。

为了提供退出功能，打开 *rango/views.py* 文件，添加名为 `user_logout()` 的视图，代码如下：

```
# 使用 login_required() 装饰器限制
# 只有已登录的用户才能访问这个视图
@login_required
def user_logout(request):
    # 可以确定用户已登录，因此直接退出
    logout(request)
    # 把用户带回首页
    return HttpResponseRedirect(reverse('index'))
```

别忘了在 *views.py* 模块的顶部导入 `logout()` 函数：

```
from django.contrib.auth import logout
```

接下来，修改 Rango 应用的 *urls.py* 模块，把 URL */rango/logout/* 映射到 `user_logout()` 视图上：

```
url(r'^logout/$', views.user_logout, name='logout'),
```

用户退出机制实现了，最后再完善一下。如果页面中有个链接，点击后便退出那就好了。然而，稍微动点脑筋便会想出一个问题：有必要向未登录的用户显示退出链接吗？可能没必要，让未登录的用户方便登录或许更好。

跟之前一样，我们要修改 Rango 应用的 *base.html* 模板，通过模板上下文中的 `user` 对象判断要显示哪些链接。在文件底部找到链接列表，替换为下述代码。注意，我们还加上了 */rango/restricted/* URL 上受限制的页面。

```
<ul>
{% if user.is_authenticated %}
    <li><a href="{% url 'restricted' %}">Restricted Page</a></li>
    <li><a href="{% url 'logout' %}">Logout</a></li>
{% else %}
    <li><a href="{% url 'login' %}">Login</a></li>
    <li><a href="{% url 'register' %}">Register Here</a></li>
{% endif %}
    <li><a href="{% url 'add_category' %}">Add a New Category</a></li>
    <li><a href="{% url 'about' %}">About</a></li>
    <li><a href="{% url 'index' %}">Index</a></li>
</ul>
```

这段代码的意思是，已登录的用户能看到“Restricted Page”和“Logout”链接，未登录的用户能看到“Register Here”和“Login”链接。“Add a New Category”和“About”不在条件判断语句块中，因此不管用户是否登录都能看到。

9.10 扩展功能

本章涵盖了在 Django 应用中验证用户身份的多个重要方面。我们介绍了如何在项目中安装 Django 提供的 `django.contrib.auth` 应用，说明了如何为 `django.contrib.auth.models.User` 模型增加字段，还讲解了实现用户注册、登录、退出和访问限制等功能的详细步骤。关于用户身份验证和注册的进一步信息，请阅读 Django 文档。

很多 Web 应用对用户身份要求的更严。例如，注册用户时会做多种安全检查，确保提供的电子邮件地址是有效的。我们可以自己实现这个功能，但是既然有现成的方案，为什么要重造轮子呢？`django-registration-redux` 应用能极大地简化为用户身份验证添加额外功能的过程。详情参见第 11 章。

</> 练习

请完成以下练习，巩固本章所学的知识。

- 改进 Rango 应用，只让已登录的用户添加分类和网页，而未登录的用户只能查看分类和网页。还要确保只有已登录的用户才能看到“Add a Page”链接。
- 用户输入错误的用户名或密码时显示有用的错误消息。
- 使用模板重构限制访问内容那个页面。把模板命名为 *restricted.html*。这个模板同样继承 Rango 应用的 *base.html* 模板。

10 cookie 和会话

本章介绍会话处理和 cookie 存储的基础知识。会话和 cookie 是互不可分的两个概念，对如今的 Web 应用极为重要。前一章涉及的登录和退出功能就是基于会话和 cookie 实现的。然而，具体过程被 Django 框架隐藏了。本章将探讨这背后到底发生了什么，以及 cookie 还能用来做什么。

10.1 cookie 无处不在

Web 服务器收到请求后会返回所请求页面的内容。除了内容之外，还会返回一些 cookie。cookie 可以理解为服务器发给客户端的少量信息。准备发送请求时，客户端检查有没有与服务器地址匹配的 cookie，如果有，随请求一起发送。服务器收到请求后把 cookie 放在请求的上下文中解释，然后生成合适的响应。

以使用用户名和密码登录网站为例。通过身份验证后，服务器可能会把一个包含用户名的 cookie 发给浏览器，指明那个用户已经登录网站。随后的请求会连同这个信息一起发给服务器，从而渲染针对已登录用户的页面，例如在页面的某个位置显示用户名。然而，会话不是永久的，因为 cookie 不会一直存在，而会在某一时刻过期。涉及敏感信息的 Web 应用可能会在几分钟之后就让 cookie 过期。要求没这么严格的 Web 应用可能会在半小时之后，甚至几周之后让 cookie 过期。

★ cookie 缘起 ★

cookie 这个词不是取自食物，[1] 而是来自“magic cookie”（收到后不经修改就发送的数据包）。1994 年，MCI 向 Netscape Communications 提交了一个提案，描述了一种跨 HTTP 请求的持久存储方式。这个提案源自他们开发的电子商务解决方案需要一种可靠的方式存储用户购物篮中的内容。Netscape 的程序员 Lou Montulli 实现了这个提案，将其命名为 magic

1. 英文单词 cookie 有“饼干”意。——译者注

cookie。

关于 cookie 及其历史的更多信息，请阅读维基百科。当然，这个伟大提议是有软件专利的，详情参见 Montulli 在美国申请的 5774670 号专利。

以 cookie 的形式传递信息可能会导致 Web 应用存在安全漏洞，因此 Web 应用的开发者在使用 cookie 时一定要谨慎。想使用 cookie 时，一定要问自己：真的有必要把这个信息存储在客户端的 cookie 中吗？很多情况下，有更为安全的方案。比如说，电商网站通过 cookie 传送用户的信用卡卡号就极不安全。试想，如果用户的电脑被攻击了呢？恶意程序可能会攫取 cookie，致使用户的信用卡卡号泄露——这全是因为 Web 应用的设计有缺陷。本章将介绍客户端 cookie 和服务器端会话存储的基础知识。

■ 欧盟对 cookie 的要求 ■

2011 年，欧盟发布了一部关于 cookie 的法律。根据此部法律，存贮在欧盟境内的网站，在用户首次访问时应该显示一个关于使用 cookie 的提醒消息。图 10-1 是 BBC 新闻网站显示的提醒消息。

对开发网站的你来说，一定要知道这部法律，以及其他与可访问性有关的法律。

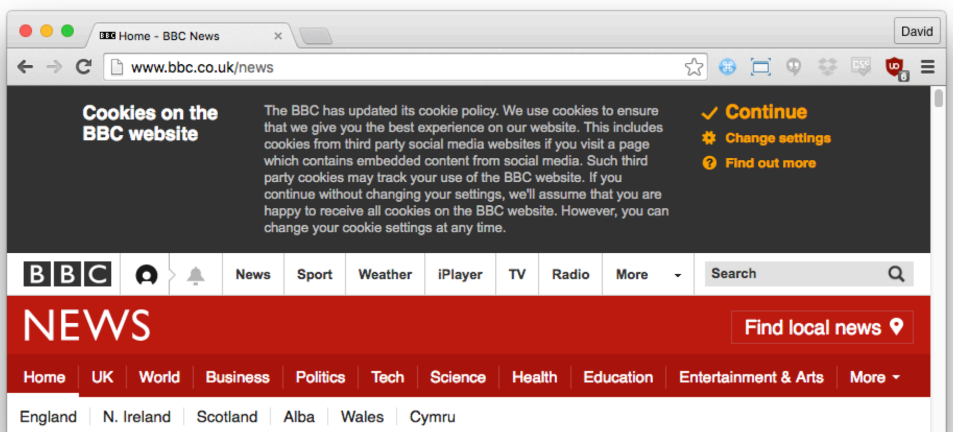


图 10-1：BBC 新闻网站（存贮在英国）顶部显示的 cookie 提醒消息

10.2 会话和无状态协议

Web 浏览器（客户端）与服务器之间的通信都借由 HTTP 协议。前面说过，HTTP 是无状态的协议。因此，Web 浏览器所在的客户端电脑每次请求服务器中的资源（HTTP GET）或者把资源发给服务器（HTTP POST）都要建立新的网络连接（TCP 连接）。[2]

由于客户端和服务器之间无法建立持久连接，所以两端的软件都不能只依靠连接维持会话状态。例如，客户端发送的每个请求都要指明哪个用户在当前电脑上已登录 Web 应用。这是客户端与服务器之间的一种对话，是半永久性信息交互机制（即会话）的基础。鉴于 HTTP 是无状态的协议，想维持会话状态还是有一定难度的，不过我们无需担心，现在有多种技术能解决这个问题。

维持状态最常用的一种方式是在客户端电脑的 cookie 中存储会话 ID。你可以把会话 ID 理解为一种令牌（一串字符，或一个字符串），我们通过它唯一标识 Web 应用中的会话。这种方式无需在客户端的 cookie 中存储大量信息（例如用户名、姓名或密码），存储的只是会话 ID。通过这个 ID 可以从 Web 服务器中获取包含所需信息的数据结构。通过这种方式存储用户的信息更安全，不会由于客户端的漏洞或者连接被监听而导致信息泄露。

现代的浏览器，只要不特意关闭，都支持 cookie。你访问的几乎每个网站都会为你创建一个会话。不信的话现在你就可以确认，如图 10-2 所示。以 Google Chrome 为例，通过开发者工具便可以查看当前访问网站的 cookie。依次点击菜单“Chrome 设置 > 更多工具 > 开发者工具”，在打开的开发者工具面板中点击“Application”标签页，在左侧的“Storage”菜单中找到“Cookies”。如果你打开的是 Rango 应用的某个页面，应该会看到一个名为 `sessionid` 的 cookie。这个 cookie 的值是一系列字母和数字，Django 就是通过这个值唯一标识你电脑中的这个会话的。通过会话 ID 可以获取所有会话信息，只不过这些信息存储在服务器端。

★ 不使用 cookie 的方式 ★

持久存储状态信息的另一种方式是在 URL 中编码会话 ID。例如，你可能见过这样的 PHP 网页地址：*http://www.site.com/index.php?sessid=someseeminglyrandomandlongstring1234*。这种方式不在客户端设备中存储 cookie，但是 URL 不太好看。这与 Django 推崇的人类可读的简洁 URL 相悖。

---

2. HTTP 1.1 其实支持在一个 TCP 网络连接中发送多个请求。这样能极大地提升性能，尤其是在延迟高的网络连接中（例如传统的拨号上网或卫星上网）。这种技术称为 HTTP 管线化，详情参见维基百科。

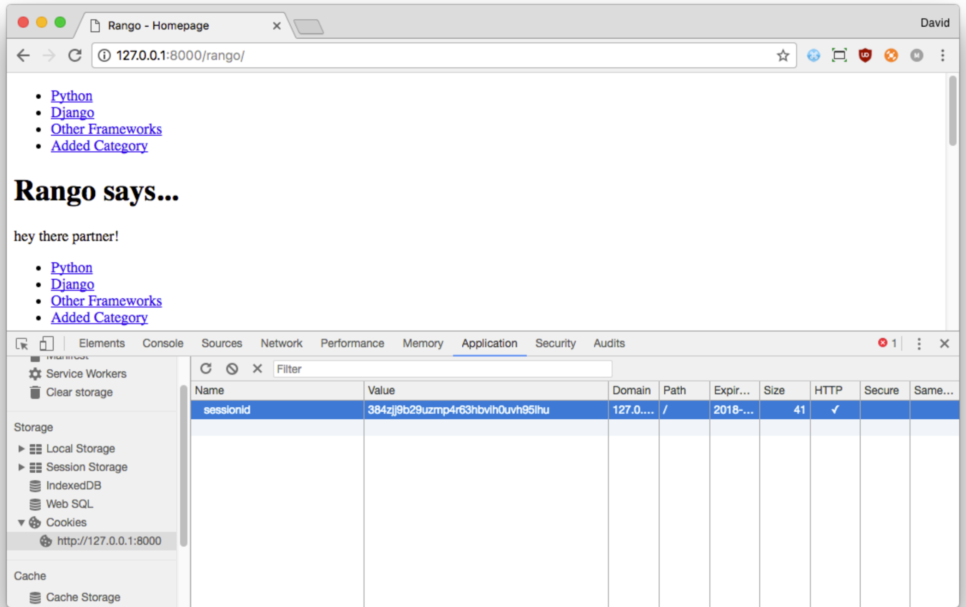


图 10-2：在 Google Chrome 的开发者工具中查看 sessionid cookie

10.3 在 Django 中设置会话

虽然你需要的功能可能已经设置好，拿来就能用，但最好知道 Django 的某个模块提供的是什么功能。本章所讲的会话，在 Django 中通过中间件实现。

为了确保万无一失，请打开 Django 项目的 *settings.py* 文件，找到 `MIDDLEWARE` 列表。这个列表中应该有个元素是 `django.contrib.sessions.middleware.SessionMiddleware`。如果没有，请自己动手加上。`sessionid` cookie 就是由 `SessionMiddleware` 中间件创建的。

`SessionMiddleware` 中间件支持多种存储会话信息的方式，可以存在文件中、数据库中，甚至是内存中。最简单的方法是使用 `django.contrib.sessions` 应用，把会话信息存储在模型/数据库中（模型为 `django.contrib.sessions.models.Session`）。若想使用这种方法，Django 项目的 `INSTALLED_APPS` 设置中还要列出 `django.contrib.sessions`。别忘了，添加这个应用后还要使用迁移命令更新数据库。

★ 把会话存入缓存 ★

如果想提升性能，可以考虑使用某种缓存机制存储会话信息。详情参见 Django 文档。

10.4 测试是否支持 cookie

虽然所有现代的 Web 浏览器都支持 cookie，但是在某些安全级别下可能会被禁用。因此，在使用 cookie 之前要测试一下。不过，这一步基本上是多余的。

测试是否支持 cookie 时，可以使用 Django 的 `request` 对象提供的三个便利方法：`set_test_cookie()`、`test_cookie_worked()` 和 `delete_test_cookie()`。我们要在一个视图中设定测试 cookie，然后在另一个视图中检查那个 cookie 是否存在。注意，必须使用两个不同的视图，因为我们要等到下一次请求才能确认客户端是否接受服务器发送的 cookie。

这里，我们将使用两个现有的视图做个简单的测试：`index()` 和 `about()`。但是我们不在页面中显示什么内容，而是通过 Django 开发服务器在终端里的输出判断是否支持 cookie。

打开 Rango 应用的 *views.py* 文件，找到 `index()` 视图。在视图中添加下述代码。为了确保这一行一定会被执行，请将其放在视图的第一行。

```
request.session.set_test_cookie()
```

然后找到 `about()` 视图，把下面三行添加到函数前部：

```
if request.session.test_cookie_worked():
    print("TEST COOKIE WORKED!")
    request.session.delete_test_cookie()
```

最后，启动 Django 开发服务器，在浏览器中访问 Rango 应用的首页，接着再访问关于页面。在 Django 开发服务器的控制台中应该会看到“TEST COOKIE WORKED!”，如图 10-3 所示。

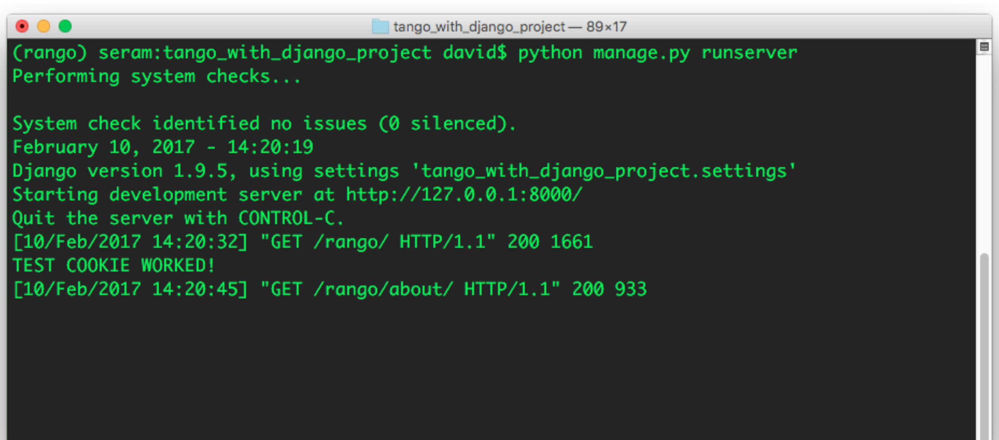


图 10-3：Django 开发服务器的控制台中显示有“TEST COOKIE WORKED!”消息

如果没显示上述消息，请检查浏览器的安全设置。某些设置可能会导致浏览器不接受 cookie。

10.5 客户端 cookie：访问次数统计示例

知道 cookie 的工作原理之后，下面来实现一个非常简单的网站访问次数统计功能。为此，我们要创建两个 cookie：一个记录用户访问 Rango 应用的次数，另一个记录用户最后一次访问网站的时间。之所以记录最后访问时间，是因为我们只想一天增加一次访问次数，以防有人恶意刷次数。

假设有用户访问 Rango 应用的合理位置是首页。我们要定义一个函数，传入 `request` 和 `response` 对象，让它处理 cookie。然后在 Rango 应用的 `index()` 视图中调用。打开 Rango 应用的 *views.py* 文件，添加下述函数。注意，严格来说这不是视图函数，而是个辅助函数，因为它不返回 `response` 对象。

```
def visitor_cookie_handler(request, response):
    # 获取网站的访问次数
    # 使用 COOKIES.get() 函数读取“visits”cookie
    # 如果目标 cookie 存在，把值转换为整数
    # 如果目标 cookie 不存在，返回默认值 1
    visits = int(request.COOKIES.get('visits', '1'))

    last_visit_cookie = request.COOKIES.get('last_visit', str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                        '%Y-%m-%d %H:%M:%S')

    # 如果距上次访问已超过一天……
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        # 增加访问次数后更新“last_visit”cookie
        response.set_cookie('last_visit', str(datetime.now()))
    else:
        # 设定“last_visit”cookie
        response.set_cookie('last_visit', last_visit_cookie)

    # 更新或设定“visits”cookie
    response.set_cookie('visits', visits)
```

这个辅助函数的参数有两个，分别为 `request` 和 `response` 对象，因为我们既需要从入站请求中读取 cookie，也需要把 cookie 添加到响应中。在这个函数中，我们调用了 `request.COOKIES.get()` 函数，这也是一个辅助函数，由 Django 提供。如果指定的 cookie 存在，`COOKIES.get()` 函数返回

cookie 的值；如果不存在，我们可以提供一个默认值。

得到两个 cookie 的值之后，计算两次访问的间隔有没有超过一天。如果无需相隔一天，可以把 `.days` 改成 `.seconds`，这样只要相差一秒就能更新访问次数。

注意，所有 cookie 的值都是字符串。不要以为存储整数的 cookie 会返回整数类型的值。你要自行转换为正确的类型，因为 cookie 并不知道它存储的值是什么类型。

如果 cookie 不存在，可以在 `response` 对象上调用 `set_cookie()` 方法，创建一个 cookie。这个方法接受两个参数，一个是想创建的 cookie 名称（字符串形式），另一个是 cookie 的值。传入的 cookie 值不限类型，`set_cookie()` 方法会自动将其转换为字符串。

这个函数用到了 `datetime` 模块，因此要在 *views.py* 文件的顶部将其导入。

```
from datetime import datetime
```

然后更新 `index()` 视图，调用 `visitor_cookie_handler()` 辅助函数。为此，要先提取得到 `response` 对象。

```
def index(request):
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]
    context_dict = {'categories': category_list, 'pages': page_list}

    # 提前获取 response 对象，以便添加 cookie
    response = render(request, 'rango/index.html', context_dict)

    # 调用处理 cookie 的辅助函数
    visitor_cookie_handler(request, response)

    # 返回 response 对象，更新目标 cookie
    return response
```

现在，访问 Rango 应用的首页，然后打开浏览器中的 cookie 查看工具（例如 Google Chrome 的开发者工具），你应该能看到 `visits` 和 `last_visit` 两个 cookie，如图 10-4 所示。此外，还可以更新 *index.html* 模板，添加 `<p>visits: {{ visits }}</p>`，显示访问次数。

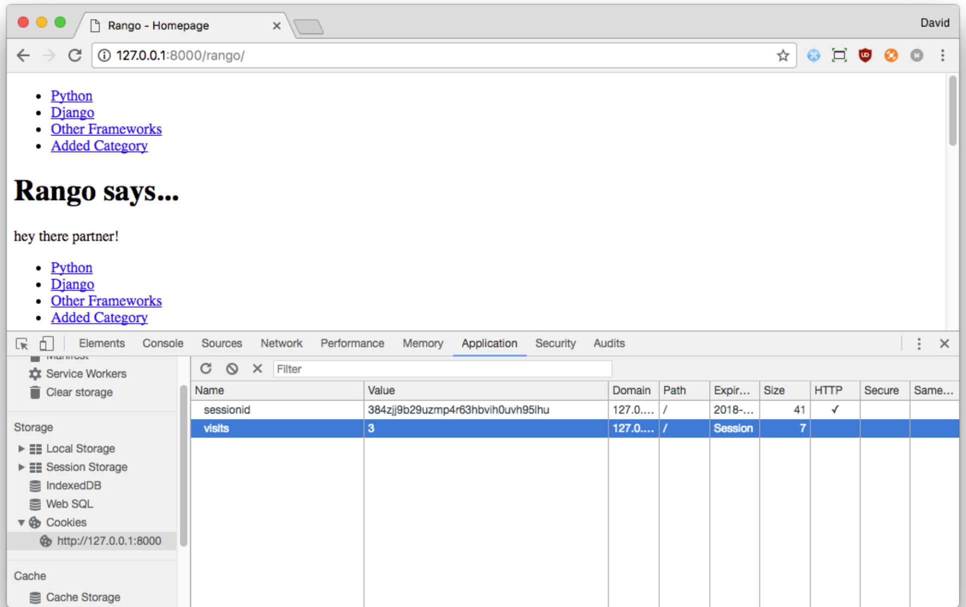


图 10-4：在 Google Chrome 中打开开发者工具查看 Rango 应用的 cookie；留意 visits cookie，其值表明一共访问了三次，而且每次至少相隔一天

10.6 会话数据

前一节的示例展示了如何存储和处理客户端 cookie，此时数据存储在客户端。然而，为了会话信息的安全，最好将其存储在服务器端，然后通过存储在客户端的会话 ID 访问会话数据。

使用基于会话的 cookie 的基本步骤如下：

1. 确保 *settings.py* 模块中的 `MIDDLEWARE_CLASSES` 列表里有 `django.contrib.sessions.middleware.SessionMiddleware`。
2. 配置会话后端。确保 *settings.py* 模块中的 `INSTALLED_APPS` 列表里有 `django.contrib.sessions`。如果没有，加上，然后运行数据库迁移命令 `python manage.py migrate`。
3. 默认使用的是数据库后端，不过也可以设置为其他后端（例如缓存）。详情参见 Django 文档。

基于会话的 cookie 不直接存储在请求中（因此也不保存在客户端设备中），读取使用 `request.session.get()` 方法，存储新值使用 `request.session[]`。注意，为了记住客户端设备，

依然要在客户端 cookie 中存储会话 ID。然而，用户/会话数据全部保存在服务器端。Django 提供的会话中间件能处理这两端的操作。

为了使用服务器端存储的数据，我们要重构前一节编写的代码。首先，更新 `visitor_cookie_handler()` 函数，改为从服务器端存取 cookie：调用 `request.session.get()` 方法读取 cookie，通过 `request.session[]` 更新 cookie。简便起见，我们定义一个辅助函数，名为 `get_server_side_cookie()`，让它从请求中读取 cookie，如果会话数据中有指定的 cookie，返回其值，否则返回默认值。

既然现在所有 cookie 都存储在服务器端，我们就不用直接修改响应了。鉴于此，`visitor_cookie_handler()` 函数中的 `response` 可以删掉了。

```
# 辅助函数
def get_server_side_cookie(request, cookie, default_val=None):
    val = request.session.get(cookie)
    if not val:
        val = default_val
    return val

# 更新后的函数定义
def visitor_cookie_handler(request):
    visits = int(get_server_side_cookie(request, 'visits', '1'))
    last_visit_cookie = get_server_side_cookie(request,
                                               'last_visit',
                                               str(datetime.now()))
    last_visit_time = datetime.strptime(last_visit_cookie[:-7],
                                        '%Y-%m-%d %H:%M:%S')

    # 如果距上次访问已超过一天……
    if (datetime.now() - last_visit_time).days > 0:
        visits = visits + 1
        # 增加访问次数后更新“last_visit”cookie
        request.session['last_visit'] = str(datetime.now())
    else:
        # 设定“last_visit”cookie
        request.session['last_visit'] = last_visit_cookie

    # 更新或设定“visits”cookie
    request.session['visits'] = visits
```

更新完处理 cookie 的辅助函数之后，接下来要更新 index() 视图。首先，把 visitor_cookie_handler(request, response) 改成 visitor_cookie_handler(request)。然后，添加下面这行，把访问次数传入上下文字典。

```
context_dict['visits'] = request.session['visits']
```

这一行要在调用 render() 函数之前执行，否则无效。修改后的 index() 视图如下所示。

```
def index(request):
    request.session.set_test_cookie()
    category_list = Category.objects.order_by('-likes')[:5]
    page_list = Page.objects.order_by('-views')[:5]
    context_dict = {'categories': category_list, 'pages': page_list}

    visitor_cookie_handler(request)
    context_dict['visits'] = request.session['visits']

    response = render(request, 'rango/index.html', context=context_dict)
    return response
```

在重启 Django 开发服务器之前先把客户端现有的 cookie 删除。详情参见下面的提醒。

◆ 避免 cookie 混淆 ◆

转用基于会话的 cookie 之前强烈建议先把客户端 cookie 删除。可以在浏览器的 cookie 查看工具中一个一个删除，也可以直接清除浏览器的缓存（确保选中删除 cookie 的选项）。

★ cookie 的数据类型 ★

把会话数据存储在服务器端的额外好处是，可以把字符串数据转换成所需的类型。不过这只对内置类型有效，例如 int、float、long、complex 和 boolean。如果想存储字典或其他复杂的类型，可以考虑序列化对象。

10.7 浏览器存续期会话和持久会话

Django 的会话框架设定的会话分为浏览器存续期会话和持久会话两种。

- 浏览器存续期会话在用户关闭浏览器后过期
- 持久会话不在浏览器关闭后过期，而是由你自己指定过期时间，可以是半个小时，甚至几个月

默认情况下，浏览器存续期会话是禁用的。若想启用，打开 Django 项目的 *settings.py* 模块，添加 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 变量，把值设为 `True`。

默认启用的是持久会话，此时 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 的值为 `False`，或者不存在。持久会话还有个设置，`SESSION_COOKIE_AGE`，用于设定 cookie 的存活期。这个设置的值是一个整数，表示 cookie 存活的秒数。例如，如果把值设为 `1209600`，网站的 cookie 将在两周（14 天）后过期。

可用选项的详细说明参阅 Django 文档。Eli Bendersky 写的这篇博客文章也值得一读。

10.8 清理会话数据库

会话不断增多，存储会话信息的数据存储器随之不断增大。如果使用数据库存储 Django 会话，要定期清理数据库。使用的命令是 `python manage.py clearsessions`。Django 文档建议通过 cron 作业每天运行一次这个命令。如若不然，随着用户数量的增多，你会发现应用的性能每况愈下。

10.9 注意事项和基本流程

在 Django 应用中使用 cookie 时要注意以下几点：

- 首先，确定你的 Web 应用需要哪种 cookie。你想存储的信息需要在浏览器关闭后留存吗，能在会话结束后弃之不用吗？
- 审慎决定要在 cookie 中存储哪些信息。记住，cookie 中的信息保存在客户端电脑中，这隐藏着巨大的安全隐患，毕竟你对用户电脑的安全保护措施一无所知。涉及敏感信息时，应该考虑使用服务器端会话。
- 用户可能会把浏览器的安全设置设为较高的级别，禁止使用 cookie，从而导致网站的功能失效。一定要考虑这种情况，因为你对用户的浏览器设置没有控制权。

如果决定使用客户端 cookie，遵照下述步骤操作：

1. 必须先检查想使用的 cookie 是否存在。`request.COOKIES.has_key('<cookie_name>')` 函数返回一个布尔值，指明名为 `<cookie_name>` 的 cookie 是否存在于客户端电脑中。

❷ 如果目标 cookie 存在，便可以读取其值：`request.COOKIES[]`。`COOKIES` 属性的值是个字典，因此想读取 cookie 时，把 cookie 的名称（字符串形式）传入方括号中。记住，cookie 的值始终为字符串，不管存储的值是什么语义。因此，要做好转换类型的准备（例如使用 `int()` 或 `float()`）。

❸ 如果目标 cookie 不存在，或者想更新 cookie，把想保存的值传给生成的响应。要调用的函数是 `response.set_cookie('<cookie_name>', value)`，第一个参数是 cookie 的名称，第二个参数是要设定的值。

如果对安全要求更高，应该使用基于会话的 cookie。

❶ 首先，确保 Django 项目的 *settings.py* 模块中的 `MIDDLEWARE_CLASSES` 列表里有 `django.contrib.sessions.middleware.SessionMiddleware`。如果没有，自行加上。

❷ 使用 `SESSION_ENGINE` 配置会话后端。不同的后端配置参见 Django 文档。

❸ 通过 `requests.sessions.get()` 检查 cookie 是否存在。

❹ 通过会话字典更新或设定 cookie：`requests.session['<cookie_name>']`。

</> 练习

请完成以下练习，巩固本章所学的知识。

- 确认你的 cookie 存储在服务器端。清空浏览器的缓存和 cookie，然后确认浏览器中没有 `last_visit` 和 `visits` 两个 cookie。注意，`sessionid` cookie 应该还有。Django 通过这个 cookie 从数据库中检索存储在服务器端的会话数据。
- 更新关于页面的视图和模板，告诉访客他们访问了多少次网站。记得先调用 `visitor_cookie_handler()` 函数，再从 `request.session` 字典中读取 `visits` cookie。如若不然，`visits` cookie 不存在的话，程序会抛出错误。

11 使用 Django-Registration-Redux

在第 9 章，我们自己动手创建视图、模板和 URL 映射，为 Rango 应用添加了登录和注册功能。然而，这些功能在 Web 应用中太常见了，因此别的开发者已经创建了众多拿来即用的扩展应用，能为 Django 项目提供登录、注册、一步和两步身份验证、密码修改、密码重设等功能。本章将使用 `django-registration-redux` 包实现这些功能。

为此，我们要重构代码，删掉前面实现的登录和注册功能，然后配置项目，使用 `django-registration-redux` 应用。读完本章后，你将掌握使用外部应用的方法，一窥整个过程是多么简单。

11.1 安装和设置

首先，使用 `pip` 在你的虚拟环境中安装 `django-registration-redux` 1.4 版：

```
$ pip install -U django-registration-redux==1.4
```

然后告诉 Django，我们想使用这个应用。打开 *settings.py* 文件，更新 `INSTALLED_APPS` 列表：

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rango',
    'registration'  # 增加 registration 包
    ]
```

既然打开 *settings.py* 文件了，顺便添加这个包所需的几个配置变量：

```
# 设为 True，允许用户注册
REGISTRATION_OPEN = True
# 留一周的激活时间；当然，也可以设为其他值
ACCOUNT_ACTIVATION_DAYS = 7
# 设为 True，注册后自动登录
REGISTRATION_AUTO_LOGIN = True
# 登录后呈现给用户的页面
LOGIN_REDIRECT_URL = '/rango/'
# 未登录以及访问需要验证身份的页面时重定向的页面
LOGIN_URL = '/accounts/login/'
```

接下来，打开 *tango_with_django_project/urls.py* 文件，更新 `urlpatterns`，引入 `registration` 包的 URL 映射：

```
url(r'^accounts/', include('registration.backends.simple.urls')),
```

`django-registration-redux` 包提供了不同的注册后端，能满足不同的需求。例如，你可能想使用两步验证，向用户发送一封确认邮件，里面有个确认链接。这里，我们使用的是简单的一步注册过程，输入用户名、电子邮件地址和密码即可注册，而且成功注册后自动登录。

11.2 各项操作的 URL 映射

`django-registration-redux` 包提供了多种操作。`registration.backend.simple.urls` 中包含下述映射：

- 注册 → */accounts/register/*
- 注册完成 → */accounts/register/complete/*
- 登录 → */accounts/login/*
- 退出 → */accounts/logout/*
- 修改密码 → */password/change/*
- 重设密码 → */password/reset/*

`registration.backends.default.urls` 还提供了两步注册过程中激活账户这一步：

- 激活 → *activate/<activation_key>/*

- 成功激活 → *activate/complete/*
- 激活电子邮件
    - 激活邮件的正文（一个文本文件）
    - 激活邮件的主题（一个文本文件）

下面是关键所在。虽然 `django-registration-redux` 包提供了上述丰富的功能，但是没有提供模板，因为每个应用都有自己的结构和外观设计。所以，接下来我们要创建各视图的模板。

11.3 创建模板

`django-registration-redux` 包的快速入门指南简要说明了需要哪些模板，但是没有明确指明各模板的内容。你可以自行试验，写出所需的代码。不过我们可以走个捷径，参考 Anders Hofstee 编写的模板。

首先，在 *templates* 目录中新建一个目录，命名为 *registration*。`django-registration-redux` 应用相关的模板都保存在这个目录中。

登录页面的模板

在 *templates/registration* 目录中新建 *login.html* 文件，写入下述代码：

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Login</h1>
    <form method="post" action=".">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Log in" />
        <input type="hidden" name="next" value="{{ next }}" />
    </form>
    <p>
        Not  a member?
        <a href="{% url 'registration_register' %}">Register</a>
    </p>
{% endblock %}
```

注意，所有 URL 都使用 `url` 模板标签引用。如果记不得，可以访问 *http://127.0.0.1:8000/ac-*

counts/，这个页面列出了全部 URL 映射，以及各 URL 对应的名称（假设 *settings.py* 中设定了 `DEBUG=True`）。

注册页面的模板

在 *templates/registration* 目录中新建 *registration_form.html* 文件，写入下述代码：

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Register Here</h1>
    <form method="post" action=".">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Submit" />
    </form>
{% endblock %}
```

注册完成页面的模板

在 *templates/registration* 目录中新建 *registration_complete.html* 文件，写入下述代码：

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Registration Complete</h1>
    <p>You are now registered</p>
{% endblock %}
```

退出页面的模板

在 *templates/registration* 目录中新建 *logout.html* 文件，写入下述代码：

```
{% extends "rango/base.html" %}
{% block body_block %}
    <h1>Logged Out</h1>
    <p>You are now logged out.</p>
{% endblock %}
```

试一下注册过程

启动 Django 开发服务器，访问 *http://127.0.0.1:8000/accounts/register/*。注意，注册表单中有两个密码字段，用于确认密码。自己注册试试，两次输入不同的密码。

这只是整个过程的其中一步。

重构项目

接下来更新 *base.html* 模板，使用新的 URL 和视图。

- 把注册链接改为 `<a href="{% url 'registration_register' %}">`。
- 把登录链接改为 `<a href="{% url 'auth_login' %}">`。
- 把退出链接改为 `<a href="{% url 'auth_logout' %}?next=/rango/">`。
- 在 *settings.py* 文件中，把 `LOGIN_URL` 改为 `'/accounts/login/'`。

注意，退出 URL 的后面有 `?next=/rango/`，这样退出后才会重定向到 Rango 应用的首页。否则，退出后将重定向到退出页面，这样不太友好。

接下来要停用 Rango 应用中的注册、登录和退出功能：把相关的 URL 映射、视图和模板删除（或者注释掉）。

修改注册流程

现在，用户成功注册后会转到注册完成页面。这样感觉有点多此一举。因此，我们修改一下，转向首页。为此要覆盖 `registration.backends.simple.views` 提供的 `RegistrationView`。打开 *tango_with_django_project/urls.py* 文件，导入 `RegistrationView`，然后定义一个 `RegistrationView` 的子类。

```
from registration.backends.simple.views import RegistrationView

# 定义一个类
# 用户成功注册后重定向到首页
class MyRegistrationView(RegistrationView):
    def get_success_url(self, user):
        return '/rango/'
```

然后修改 Django 项目的 *urls.py* 模块中的 `urlpatterns` 列表，在 `accounts` 模式前添加下述代码，

以便在其他 *accounts/* URL 之前匹配 *accounts/register*。注意，修改的不是 `rango` 目录中的 *urls.py* 模块。

```
url(r'^accounts/register/$',
    MyRegistrationView.as_view(),
        name='registration_register'),
```

这样修改之后，成功注册后便会重定向到我们指定的页面。

</> 练习和提示

- 为用户提供修改密码功能。
- 提示：参考 Anders Hofstee 编写的模板。
- 提示：修改密码的 URL 是 *accounts/password/change/*，指明密码已修改的 URL 是 *accounts/password/change/done/*。

12 集成 Bootstrap

本章使用 Twitter Bootstrap [1] 装饰 Rango 应用。Bootstrap 是最为流行的 HTML、CSS 和 JavaScript 框架，支持响应式设计，简单易用。

★ 层叠样式表 ★

> 如果你不熟悉 CSS，推荐阅读《CSS 实战手册》一书，学习层叠样式表的基础知识。

现在打开 Bootstrap 的网站，可以看到网站给出了很多示例代码，说明了各组件的用法。此外，Bootstrap 的网站中还有一些布局示例，我们可以参照这些示例设计 Rango 应用。

在众多布局中，笔者觉得这个管理后台比较适合 Rango，有导航栏、侧边栏（用于显示分类列表）和主内容区。

打开这个管理后台布局，保存 HTML 源码，存储在 *templates/rango* 目录里，命名为 *base_bootstrap.html*。

为了在 Rango 应用中使用，还要做些修改：

- 把所有 `../../` 路径替换为 `http://v4-alpha.getbootstrap.com/`。
- 把 `dashboard.css` 的路径替换为绝对地址：`http://v4-alpha.getbootstrap.com/examples/dashboard/dashboard.css`。
- 把导航栏中的搜索框删除。

1. 翻译时 Bootstrap 4 已经正式发布。原书使用的还是 Bootstrap 4 Alpha 版，因此 Bootstrap 相关的链接仍然指向 v4 Alpha 版。——译者注

- 把页面中不相关的内容都删掉，替换为 {% block body_block %}{% endblock %}。
- 把页面的标题元素改为 <title> Rango - {% block title %}How to Tango with Django!{% endblock %} </title>。
- 把网站名称（“Dashboard”）改为“Rango”。
- 在导航栏中添加首页、登录和注册等页面的链接。
- 添加侧边栏区块：{% block sidebar_block %}{% endblock %}。
- 在 DOCTYPE 下面添加 {% load staticfiles %}。

12.1 模板

◆ 不要复制粘贴 ◆

笔者早就提醒过，不要直接复制粘贴书中的代码。请自己输入，在输入的过程中想一想下述 HTML 标记的作用。如果你不知道某个标签的作用，请在网上搜索。如果你不知道某个 Bootstrap CSS 类的作用，请查阅文档。

```
<!DOCTYPE html>
{% load staticfiles %}
{% load rango_template_tags %}
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
                               initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="{% static 'images/favicon.ico' %}">
    <title>
      Rango - {% block title %}How to Tango with Django!{% endblock %}
    </title>
    <!-- Bootstrap core CSS -->
    <link href="http://v4-alpha.getbootstrap.com/dist/css/bootstrap.min.css"
        rel="stylesheet">
    <!-- Custom styles for this template -->
```

```
    <link href=
           "http://v4-alpha.getbootstrap.com/examples/dashboard/dashboard.css"
          rel="stylesheet">
</head>
<body>

<nav class="navbar navbar-toggleable-md navbar-inverse fixed-top bg-inverse">
    <button class="navbar-toggler navbar-toggler-right hidden-lg-up"
            type="button"
            data-toggle="collapse"
            data-target="#navbar"
            aria-controls="navbar"
            aria-expanded="false"
            aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <a class="navbar-brand" href="{% url 'index' %}">Rango</a>

    <div class="collapse navbar-collapse" id="navbar">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="{% url 'index' %}">
                    Home
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'about' %}">
                    About
                </a>
            </li>
            {% if user.is_authenticated %}
            <li class="nav-item">
                <a class="nav-link" href="{% url 'restricted' %}">
                    Restricted Page
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url 'add_cateory' %}">
                Add a New Category
                </a>
```

```
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'auth_logout' %}?next=/rango/">
                        Logout
                    </a>
                </li>
                {% else %}
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'registration_register' %}">
                        Register Here
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'auth_login' %}">
                        Login
                    </a>
                </li>
                {% endif %}
            </ul>
        </div>
    </nav>

    <div class="container-fluid">
        <div class="row">
        <div class="col-sm-3 col-md-2 sidebar">
            {% block sidebar_block %}
                {% get_category_list category %}
            {% endblock %}
        </div>
        <div class="col-sm-9 offset-sm-3 col-md-10 offset-md-2 main">
            {% block body_block %}{% endblock %}
        </div>
        </div>
    </div>
    <!-- Bootstrap core JavaScript
        ================================================== -->
        <!-- Placed at the end of the document so the pages load faster -->
        <script
          src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js">
        </script>
```

```
    <script
      src="http://v4-alpha.getbootstrap.com/dist/js/bootstrap.min.js">
    </script>
    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <script
      src=
    "http://v4-alpha.getbootstrap.com/assets/js/ie10-viewport-bug-workaround.js">
    </script>
  </body>
  </html>
```

创建好这个模板之后，下载 favicon，保存到 *static/images/* 目录中。

仔细看一下上述 HTML 源码，你会发现里面有好多使用 `<div>` 标签创建的结构。整个页面基本上分为两部分：`<nav>` 标签里的顶部导航栏和 `<div class="container-fluid">` 标签里的主内容区。主内容区里还有两个 `<div>` 标签，分别放置 `sidebar_block` 和 `body_block` 区块。

注意，这个 HTML 模板从外部网站引用 CSS 和 JavaScript 文件。因此，为了正确加载那些文件，需要联网。

■ 想离线工作? ■

除了从外部网站引用 CSS 和 JavaScript 文件之外，还可以下载相关文件，保存到静态文件目录中。如果你选择这么做，别忘了更新模板中对这些文件的引用地址。

12.2 调整模板

接下来还要做些调整，把 *base.html* 模板中的代码替换为 *base_bootstrap.html* 中的代码。为此，可以先把 *base.html* 中现有的代码注释掉，然后复制粘贴 *base_bootstrap.html* 中的代码。

然后刷新网页，你会发现页面变得精美多了。以关于页面为例，修改前后的对比如图 12-1 和图 12-2 所示。

浏览一下其他页面，因为都继承自基模板，所以看起来应该都不错，但也有不足之处。接下来我们逐一修改各页面的模板，使用 Bootstrap 提供的 CSS 类改进外观。

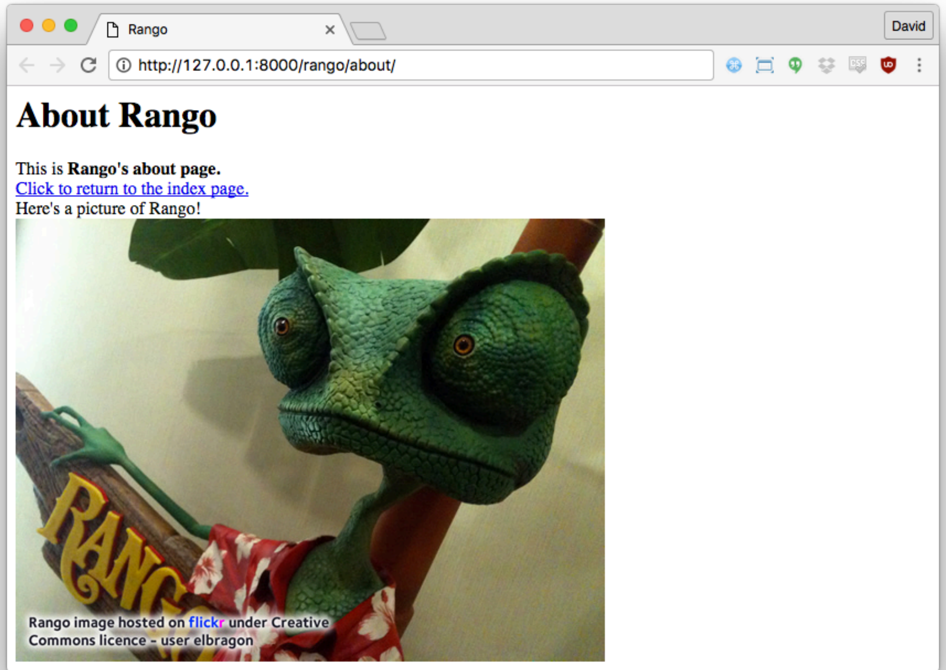


图 12-1：没有样式的关于页面

首页

首页中最受欢迎的分类和网页最好分两栏显示。浏览 Bootstrap 的示例后我们发现，Narrow Jumbotron 中的两栏就符合我们的要求。查看网页源码，找到实现分栏的 HTML 代码：

```
<div class="row marketing">
    <div class="col-lg-6">
        <h4>Subheading</h4>
        <p>Donec id elit non mi porta gravida at eget metus.
            Maecenas faucibus mollis interdum.</p>
        <h4>Subheading</h4>
    </div>
    <div class="col-lg-6">
        <h4>Subheading</h4>
        <p>Donec id elit non mi porta gravida at eget metus.
            Maecenas faucibus mollis interdum.</p>
    </div>
</div>
```

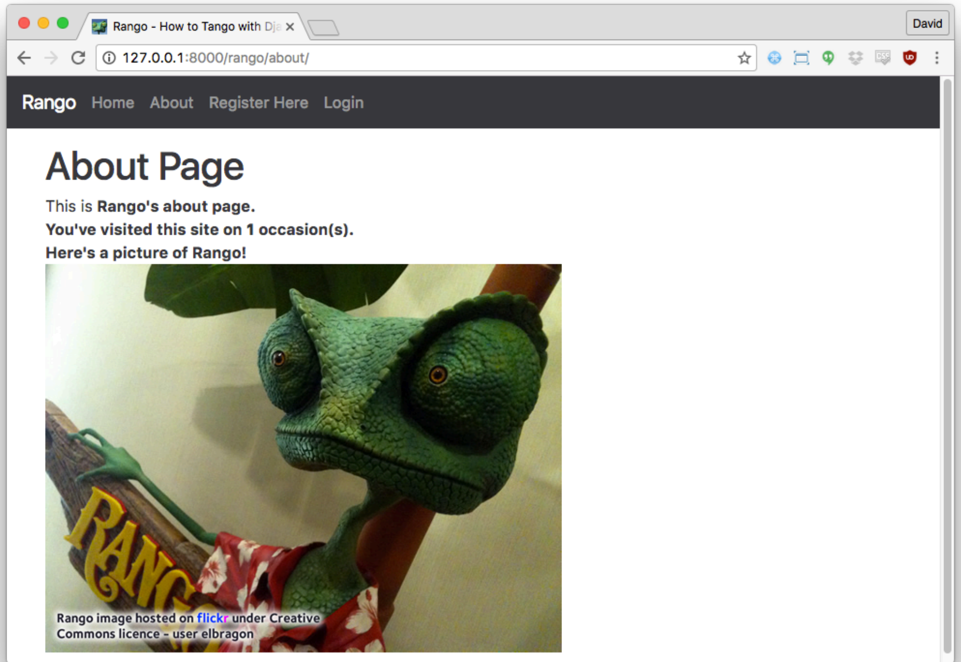


图 12-2：集成 Bootstrap 后的关于页面

`<div class="row marketing">` 中有两个类为 `col-lg-6` 的 `<div>` 元素。Bootstrap 采用栅格布局，一个容器分成 12 份。`col-lg-6` 类表明所在的分栏占 6 份，即容器（`<div class="row marketing">`）宽度的一半。

参照上述代码，修改 *index.html* 模板。

```
{% extends 'rango/base.html' %}
{% load staticfiles %}
{% block title_block %}
    Index
{% endblock %}
{% block body_block %}
<div class="jumbotron">
    <h1 class="display-3">Rango says...</h1>
    {% if user.is_authenticated %}
        <h1>hey there {{ user.username }}!</h1>
    {% else %}
        <h1>hey there partner! </h1>
    {% endif %}
```

```
    </div>
    <div class="row marketing">
        <div class="col-lg-6">
        <h4>Most Liked Categories</h4>
        <p>
        {% if categories %}
        <ul>
            {% for category in categories %}
            <li><a href="{% url 'show_category' category.slug %}">
                {{ category.name }}</a></li>
            {% endfor %}
        </ul>
        {% else %}
            <strong>There are no categories present.</strong>
        {% endif %}
        </p>
        </div>
        <div class="col-lg-6">
            <h4>Most Viewed Pages</h4>
            <p>
            {% if pages %}
            <ul>
                {% for page in pages %}
                <li><a href="{{ page.url }}">{{ page.title }}</a></li>
                {% endfor %}
                </ul>
            {% else %}
                <strong>There are no categories present.</strong>
             {% endif %}
            </p>
        </div>
    </div>
    <img src="{% static "images/rango.jpg" %}" alt="Picture of Rango" />
    {% endblock %}
```

我们把页面的标题放在 `<div class="jumbotron">` 中，以此突出显示。现在刷新页面，看起来好多了，不过列表项目还是丑。

下面使用 Bootstrap 提供的列表组样式美化一下。修改方法很简单，把 `<ul>` 元素改成 `<ul class="list-group">`，把 `<li>` 元素改成 `<li class="list-group-item">` 即可。再次刷新页面，现

在好点了吧？

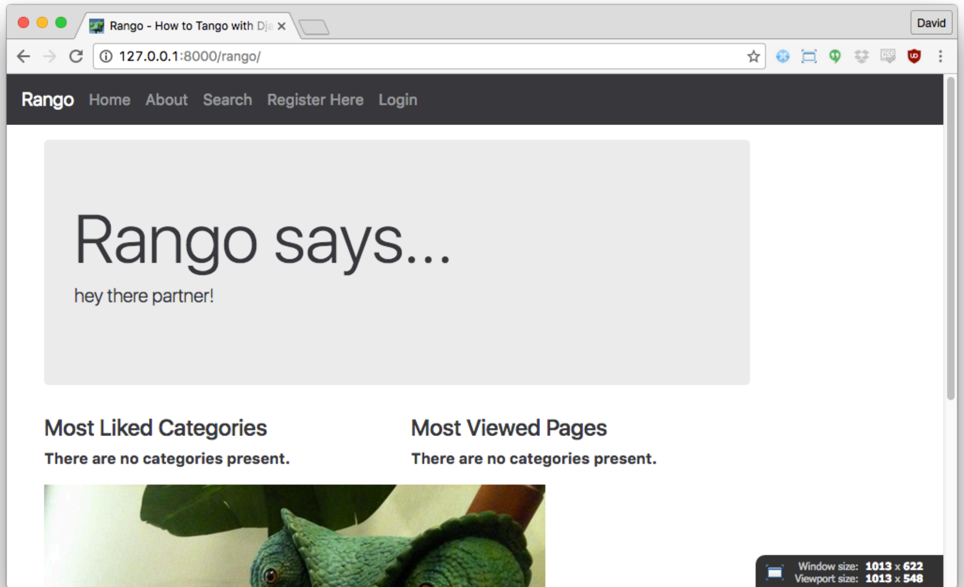


图 12-3：添加超大标题和分栏后的首页

登录页面

接下来修改登录页面。Bootstrap 网站中有个不错的登录表单。查看网页源码，你会发现我们要在现有的表单中添加几个类才能实现演示的效果。参照下述代码更新 *login.html* 模板中的 body_block 区块。

```
{% block body_block %}
<link href="http://v4-alpha.getbootstrap.com/examples/signin/signin.css"
    rel="stylesheet">
<div class="jumbotron">
    <h1 class="display-3">Login</h1>
</div>
<form class="form-signin" role="form" method="post" action=".">
    {% csrf_token %}
    <h2 class="form-signin-heading">Please sign in</h2>
    <label for="inputUsername" class="sr-only">Username</label>
    <input type="text" name="username" id="id_username" class="form-control"
           placeholder="Username" required autofocus>
```

```
        <label for="inputPassword" class="sr-only">Password</label>
        <input type="password" name="password" id="id_password" class="form-control"
               placeholder="Password" required>
        <button class="btn btn-lg btn-primary btn-block" type="submit"
                value="Submit" />Sign in</button>
    </form>
    {% endblock %}
```

除了引入 *signin.css* 样式表，修改一些 CSS 类之外，我们还删除了自动生成表单元素的代码，即 `form.as_p`。现在，各表单元素及其 `id` 都由我们自己编写。注意，表单元素的 `id` 十分重要。为了找出各表单元素的 `id`，可以查看修改之前的登录页面源码，即使用 `form.as_p` 模板标签生成的 HTML。

我们把按钮的 CSS 类设为 `btn` 和 `btn-primary`。在文档中可以看到，Bootstrap 为按钮提供了大量不同的颜色、大小和样式。

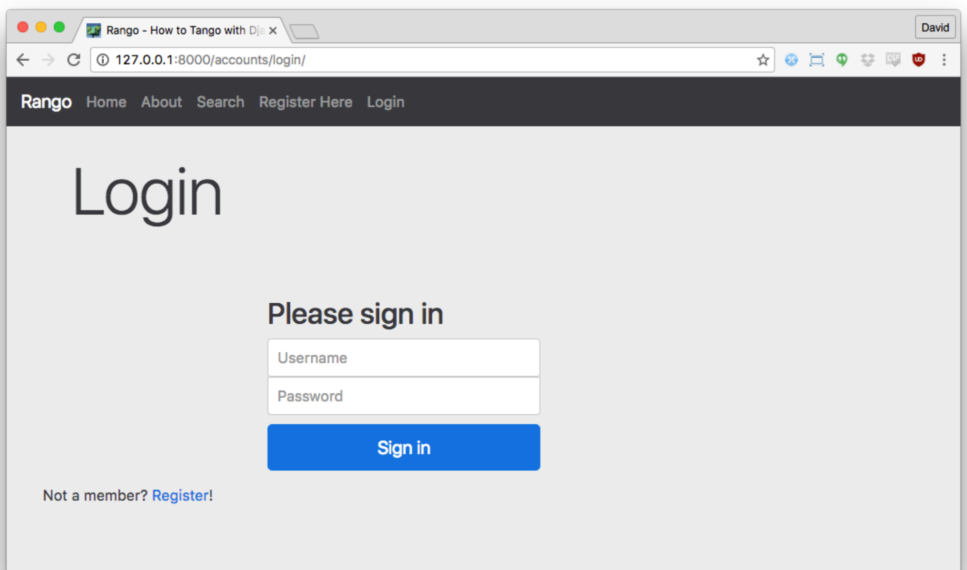


图 12-4：集成 Bootstrap 后的登录页面

其他有表单的模板

参照登录页面修改 *add_cagegory.html* 和 *add_page.html* 模板。修改后的 *add_page.html* 模板如下。

```
{% extends "rango/base.html" %}
```

```
{% block title %}Add Page{% endblock %}

{% block body_block %}
    {% if category %}
        <form role="form" id="page_form" method="post"
              action="/rango/category/{{category.slug}}/add_page/">
        <h2 class="form-signin-heading"> Add a Page to
            <a href="/rango/category/{{category.slug}}/">
                {{ category.name }}</a></h2>
        {% csrf_token %}
        {% for hidden in form.hidden_fields %}
            {{ hidden }}
        {% endfor %}
        {% for field in form.visible_fields %}
            {{ field.errors }}
            {{ field.help_text }}<br/>
            {{ field }}<br/>
        {% endfor %}
        <br/>
        <button class="btn btn-primary"
                type="submit" name="submit">
            Add Page
        </button>
        </form>
    {%  else %}
        <p>This is category does not exist.</p>
    {%  endif %}
{% endblock %}
```

</> **练习**

- 以类似的方式修改 *add_category.html* 模板。

注册页面

registration_form.html 模板中的表单可以像下面这样修改。

```
{% extends "rango/base.html" %}
{% block body_block %}

<h2 class="form-signin-heading">Sign Up Here</h2>

<form role="form"  method="post" action=".">
    {% csrf_token %}
    <div class="form-group" >
    <p class="required"><label class="required" for="id_username">
        Username:</label>
        <input class="form-control" id="id_username" maxlength="30"
             name="username" type="text" />
        <span class="helptext">
        Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.
        </span>
    </p>
    <p class="required"><label class="required" for="id_email">
        E-mail:</label>
        <input class="form-control" id="id_email" name="email"
             type="email" />
    </p>
    <p class="required"><label class="required" for="id_password1">
        Password:</label>
        <input class="form-control" id="id_password1" name="password1"
            type="password" />
    </p>
    <p class="required">
        <label class="required" for="id_password2">
            Password confirmation:</label>
        <input class="form-control" id="id_password2" name="password2"
             type="password" />
        <span class="helptext">
             Enter the same password as before, for verification.
        </span>
    </p>
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
</form>
{% endblock %}
```

同样，我们删掉了 `{{ form.as_p }}` 模板标签，各表单元素及其 CSS 类都是自己动手编写的。

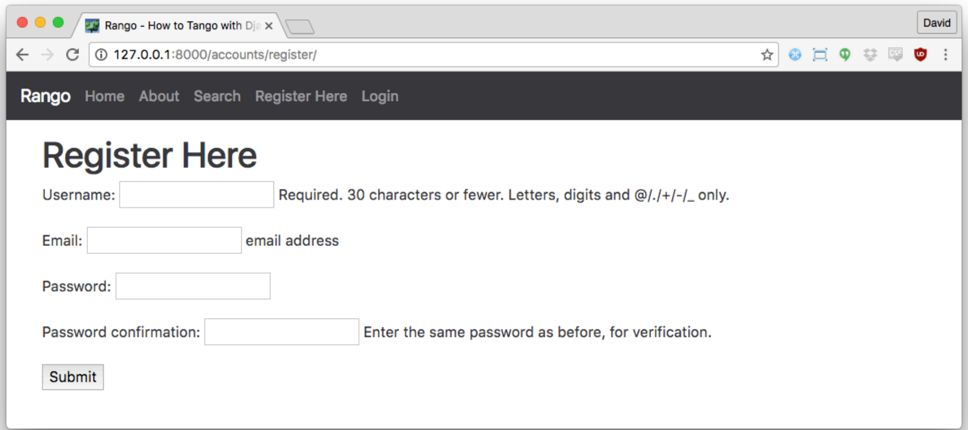


图 12-5：集成 Bootstrap 后的注册页面

◆ 蹩脚的集成方式 ◆

这不是集成 Bootstrap 的最好方式，如果能让 Django 在构建 HTML 时自动插入相关的 CSS 类就好了。

12.3 使用 Django-Bootstrap-Toolkit

除了自己动手集成之外，还可以使用 `django-bootstrap-toolkit` 这样的包。先使用 pip 安装：

```
$ pip install django-bootstrap-toolkit
```

然后打开 *settings.py* 模块，把 `bootstrap_toolkit` 添加到 `INSTALLED_APPS` 中。

为了在模板中使用这个包，首先要使用 `load` 模板标签加载它（`{% load bootstrap_toolkit %}`），然后调用一个函数，更新生成的 HTML，即 `{{ form|as_bootstrap }}`。按照下述代码更新 *add_category.html* 模板。

```
{% extends "rango/base.html" %}

{% load bootstrap_toolkit %}
{% block title %}Add Category{% endblock %}
```

```
{% block body_block %}
    <form id="category_form" method="post"
        action="{% url 'add_category' %}">
    <h2 class="form-signin-heading">Add a Category</a></h2>
    {% csrf_token %}
    {{ form|as_bootstrap }}
    <br/>
    <button class="btn btn-primary" type="submit"
         name="submit">Create Category</button>
    </form>
{% endblock %}
```

这样得到的模板更简洁，而且自动化程度更高，但是渲染结果没有手动集成的细致，因此还要做些调整和定制。

12.4 接下来

本章简要说明了如何使用 Bootstrap 装饰 Django 应用。Bootstrap 是个高度可扩展的框架，能轻易更改整体风格。StartBootstrap 网站中有很多不同的风格。

除了 Bootstrap 之外，还有很多 CSS 框架可用，例如 Zurb Foundation、Titon、Pure、GroundWorkd 和 BassCSS。

知道如何修改模板，实现响应式设计之后，下面回归正题，继续为 Rango 应用添加功能。

</> 练习

本书使用的是 Bootstrap，如果你有时间，可以试着使用其他响应式 CSS 框架装饰 Rango 应用。当然，你也可以自己设计。如果你觉得自己设计的不错，请告诉我们！

13 Webhose 搜索

至此，Rango 应用的核心功能已经实现，接下来该考虑一些高级功能了。本章将使用 Webhose API 实现搜索网页的功能，毕竟只能按分类浏览不是很方便。为此，我们需要注册 Webhose 账户，再编写一个包装脚本（wrapper），通过 API 获取查询结果。

13.1 Webhose API

Webhose 是一项在线服务，其功能是汇集众多在线源，实时整理信息。借助 Webhose API，我们能以编程的方式查询 Webhose，得到 JSON 格式的结果。返回的数据经过 JSON 解析器解释后，在应用的模板中显示出来。

Webhose 已经爬取了大量数据，但那不是我们所关注的。我们的目的是获取 Rango 应用的用户输入的词条搜索到的网页。使用 Webhose API 之前，要有 API 密钥。有了密钥，我们每个月可以免费查询一千次——这对我们的演示应用而言足够了。

★ 应用程序编程接口是什么？ ★

应用程序编程接口（Application Programming Interface，API）定义软件组件之间的交互方式。对 Web 应用来说，API 可以理解为一系列 HTTP 请求及其响应消息的结构。任何能通过互联网访问的服务都可以有 API，本章涉及的是搜索 API。如果想进一步了解 Web API，可以阅读 Luis Rei 写的这篇优秀教程。

注册 Webhose API 密钥

为了获得 Webhose API 密钥，首先要注册一个免费的 Webhose 账户。在 Web 浏览器中访问 https://www.webhose.io，点击页面右上角的“Use it for free”按钮注册。公司名称无需填写，工作邮

箱填一个有效的电子邮件地址即可。

注册好账户后，打开 Webhose 的控制面板，如图 13-1 所示。在这个页面可以看到当月已经使用的查询次数，以及还剩多少免费额度。此外，页面中还有一个图表，展示了一段时间内你查询 Webhose 的次数。向下拉滚动条，找到“Active API Key”部分。把这里显示的密钥复制到一个空文本文件中，以备后用。这是你专用的密钥，通过它对 Webhose API 的请求都算在你名下。

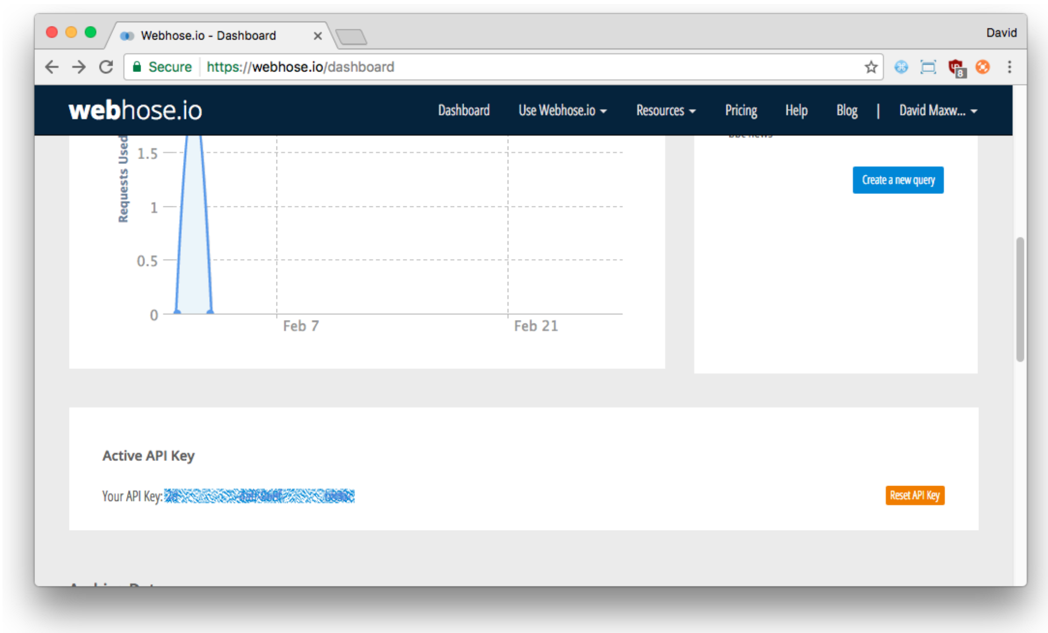


图 13-1：Webhose 的控制面板，显示 API 密钥的位置。你可能要向下拉动滚动条才能看到。上图中的 API 密钥被遮盖住了。一定要保护好自己的密钥！

记下 API 密钥后，点击顶部导航栏中的“API Playground”链接。这个页面供你试用 Webhose API 接口，你可以自己尝试一下。

1. 在“Define the query”下面的方框中输入一个词条。
2. 在“Sort By”下拉菜单中选择“Relevancy”。
3. “Crawled Since”用于设定返回什么时候爬取的结果，默认值 3 天就可以。
4. 点击“Run”按钮。稍等片刻页面中便会出现查询结果。图 13-2 是搜索“Glasgow”得到的结果。

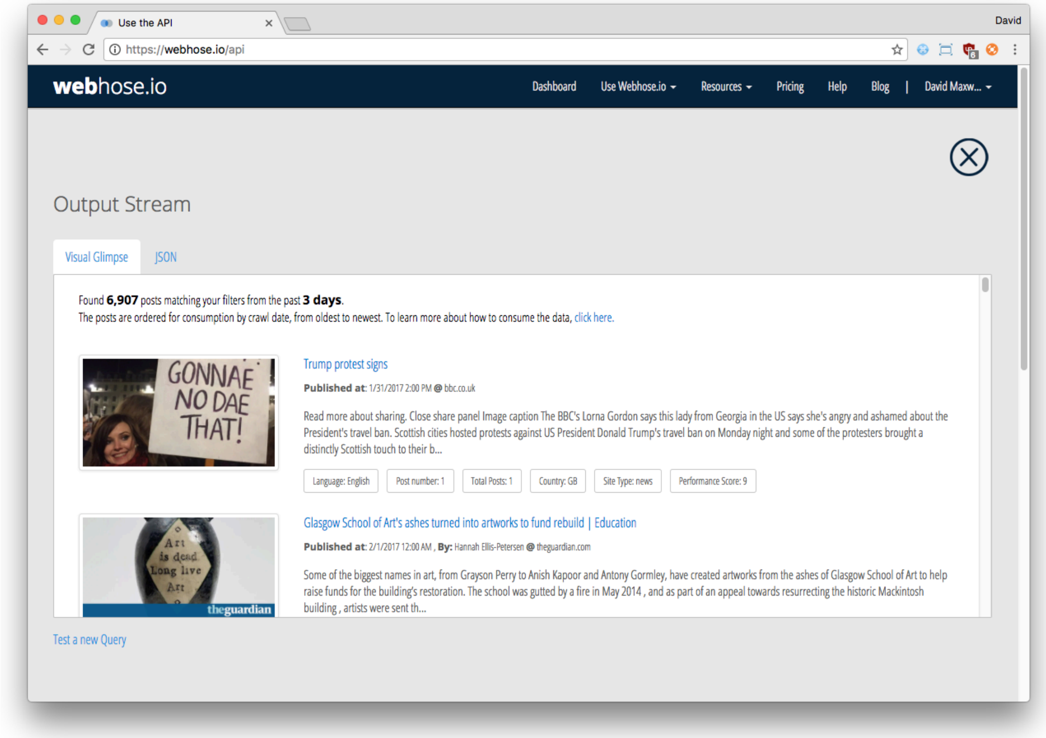


图 13-2：通过 Webhose API 查询“Glasgow”得到的响应。除了原始的 JSON 响应之外，还有预览。

看一下 Webhose API 返回的结果，以及原始的 JSON 响应。

向上拉动滚动条，找到“Endpoint”框。这里显示的 URL 就是我们的应用将要使用的查询端点。下面是一个端点 URL 示例：

```
http://webhose.io/filterWebContent?token=<KEY&format=json&sort=relevancy&q=<QUERY>
```

这个 URL 可以分为两部分，问号左边是基 URL，右边是查询字符串。基 URL 相当于 API 的路径，而查询字符串是一系列键值对（例如 `format` 是键，`json` 是值），用于告诉 API 你想做什么。在后面的示例代码中你会看到，我们将以这种方式分拆 URL，构建好查询字符串之后再合在一起，构成完整的请求 URL。

13.2 添加搜索功能

得到 Webhose API 密钥后，可以开始编写 Python 代码，向 Webhose API 发起查询请求了。在 *rango* 目录中新建一个模块（文件），命名为 *webhose_search.py*，写入下述代码。注意选择正确的 Python 版本。正如前文所说，不要盲目复制粘贴，最好自己动手输入，这样在输入的过程中你才会思考代码的作用。

★ Python 2 和 3 之间的区别 ★

Python 3 重构了 `urllib` 包，[1] 连接和处理外部 Web 资源的方式变了。下面分别给出针对 Python 2.7 和 Python 3 的代码。请根据你的环境选择正确的代码。

Python 2 版

```python
import json
import urllib
import urllib2

def read_webhose_key():
    """
    从 search.key 文件中读取 Webhose API 密钥
    返回 None（未找到密钥），或者密钥的字符串形式
    注意：把 search.key 写入 .gitignore 文件，禁止提交
    """
    # 参见 Python Anti-Patterns 小书，这是一份十分优秀的资料
    # 使用 with 打开文件
    # http://docs.quantifiedcode.com/python-anti-patterns/maintainability/
    webhose_api_key = None

    try:
        with open('search.key', 'r') as f:
            webhose_api_key = f.readline().strip()
    except:
        raise IOError('search.key file not found')

    return webhose_api_key

def run_query(search_terms, size=10):
    """
    指定搜索词条和结果数量（默认为 10），把 Webhose API 返回的结果存入列表
    每个结果都有标题、链接地址和摘要
    """
    webhose_api_key = read_webhose_key()
```

1. http://stackoverflow.com/a/2792652

```
    if not webhose_api_key:
        raise KeyError('Webhose key not found')

    # Webhose API 的基 URL
    root_url = 'http://webhose.io/search'

    # 处理查询字符串，转义特殊字符
    query_string = urllib.quote(search_terms)

    # 使用字符串格式化句法构建完整的 API URL
    # search_url 是一个多行字符串
    search_url = ('{root_url}?token={key}&format=json&q={query}'
                  '&sort=relevancy&size={size}').format(
                    root_url=root_url,
                    key=webhose_api_key,
                    query=query_string,
                    size=size)

    results = []

    try:
        # 连接 Webhose API，把响应转换为 Python 字典
        response = urllib2.urlopen(search_url).read()
        json_response = json.loads(response)

        # 迭代文章，把一篇文章作为一个字典追加到结果列表中
        # 限制摘要的长度为 200 个字符，因为 Webhose 返回的摘要可能很长
        for post in json_response['posts']:
            results.append({'title': post['title'],
                            'link': post['url'],
                            'summary': post['text'][:200]})
    except:
        print("Error when querying the Webhose API")

    # 把结果列表返回给调用方
    return results
```

Python 3 版本

```python
import json
import urllib.parse  # Py3
import urllib.request  # Py3

def read_webhose_key():
    """
    从 search.key 文件中读取 Webhose API 密钥
    返回 None（未找到密钥），或者密钥的字符串形式
    注意：把 search.key 写入 .gitignore 文件，禁止提交
    """
    # 参见 Python Anti-Patterns 小书，这是一份十分优秀的资料
    # 使用 with 打开文件
    # http://docs.quantifiedcode.com/python-anti-patterns/maintainability/
    webhose_api_key = None

    try:
        with open('search.key', 'r') as f:
            webhose_api_key = f.readline().strip()
    except:
        raise IOError('search.key file not found')

    return webhose_api_key

def run_query(search_terms, size=10):
    """
    指定搜索词条和结果数量（默认为 10），把 Webhose API 返回的结果存入列表
    每个结果都有标题、链接地址和摘要
    """
    webhose_api_key = read_webhose_key()

    if not webhose_api_key:
        raise KeyError('Webhose key not found')

    # Webhose API 的基 URL
    root_url = 'http://webhose.io/search'

```

```
    # 处理查询字符串，转义特殊字符
    query_string = urllib.parse.quote(search_terms)  # Py3

    # 使用字符串格式化句法构建完整的 API URL
    # search_url 是一个多行字符串
    search_url = ('{root_url}?token={key}&format=json&q={query}'
                  '&sort=relevancy&size={size}').format(
                    root_url=root_url,
                    key=webhose_api_key,
                    query=query_string,
                    size=size)

    results = []

    try:
        # 连接 Webhose API，把响应转换为 Python 字典
        response = urllib.request.urlopen(search_url).read().decode('utf-8')
        json_response = json.loads(response)

        # 迭代文章，把一篇文章作为一个字典追加到结果列表中
        # 限制摘要的长度为 200 个字符，因为 Webhose 返回的摘要可能很长
        for post in json_response['posts']:
            results.append({'title': post['title'],
                            'link': post['url'],
                            'summary': post['text'][:200]})
    except:
        print("Error when querying the Webhose API")

    # 把结果列表返回给调用方
    return results
```

上述代码清单实现了两个函数：`read_webhose_key()` 函数从本地文件中读取 Webhose API 密钥，`run_query()` 函数向 Webhose API 发送请求，返回得到的结果。下面分析一下这两个函数。

`read_webhose_key()`：读取 Webhose API 密钥

`read_webhose_key()` 函数从名为 *search.key* 的文件中读取 Webhose API 密钥。这个文件应该放在 Django 项目的根目录中（即 *<workspace>/tango_with_django_project/*），而不是 Rango 应用的目

录中。单独定义这样一个函数，是为了把读取和使用 API 密钥的代码分开。如果代码是公开共享的，这么做的好处就凸显出来了，毕竟我们不想让别人使用自己的 API 密钥。

现在请创建 *search.key* 文件。把之前复制的 API 密钥保存到这个文件中。这个文件只用于存储密钥，不能有任何其他内容。为了防止把这个文件提交到 GitHub 仓库，更新 *.gitignore* 文件，添加 `*.key`，排除所有扩展名为 *.key* 的文件。这样密钥就只存在于本地，不会意外提交到远程 Git 仓库中。

■ 密钥 ■

> 密钥一定要保护好，不能让他人攫取！
>
> 别让别人使用你的密钥。如果使用不当，对应的服务可能取消你的使用权限。更糟的是，如果是付费用户，你要额外支付大量费用。

`run_query()`：执行查询

`run_query()` 函数接受两个参数：`search_terms`，值为字符串，表示用户输入的搜索词条；`size`，可选，默认为 `10`，控制 Webhose API 返回的结果数量。`run_query()` 函数与 Webhose API 通信，返回一个由 Python 字典构成的列表，每个字典表示一个结果（有 `title`、`link` 和 `summary`）。上述代码清单中的注释解释了每一步操作，如果想进一步理解代码，请阅读注释。

`run_query()` 函数的逻辑大致可以分成 7 个任务，说明如下。

❶ 首先，调用 `read_webhose_key()` 函数，获取 Webhose API。

❷ 然后构建要发给 API 的查询字符串。这里要编码 URL，把特殊的字符（例如空格）转换成 Web 服务器和浏览器能理解的格式。比如，空格会转换成 `%20`。

❸ 根据 Webhose API 文档，接下来拼接编码后的 `search_terms` 字符串和 `size` 参数，以及 Webhose API 密钥，构建请求 Webhose API 的完整 URL。

❹ 然后使用 Python `urllib2`（Python 2.7.x）或 `urllib` 模块连接 Webhose API。服务器的响应存为 `response` 变量。

❺ 使用 Python `json` 库把响应转换成 Python 字典对象。

❻ 迭代字典，把 Webhose API 返回的各个结果以字典（包含 `title`、`link` 和 `summary` 键值对）为单位存入 `results` 列表。

❼ 返回 `results` 列表。

★ 研究 API 选项 ★

使用新 API 时最好研究一下文档，看有哪些选项可用。建议你研究一下 Webhose API 文档。

</> 练习

扩展 *webhose_search.py* 模块，让它可以独立运行，即可以在终端或命令提示符中运行 `python webhose_search.py`，而无需启动 Django 开发服务器。为此你要：

- 要求用户输入查询词条（使用 `raw_input()`）
- 通过 `run_query()` 发起查询，然后打印结果

把每个结果的 `title` 和 `summary` 打印出来，而且两个结果之间有换行。

此外还要修改 `read_webhose_key()` 函数，以便运行 *webhose_search.py* 脚本时能找到 *rango* 目录中的 *search.key* 文件。运行 Django 开发服务器时，Python 期望能在启动 *manage.py* 脚本的目录中找到 *search.key* 文件。直接运行 *rango* 目录中的 *webhose_search.py* 脚本时，要到上一层目录中查找 *search.key* 文件。那么应该如何修改 `read_webhose_key()` 函数才能同时在两种情况下使用呢?

如果你是在 Windows 电脑上使用 Python 2.7.x 开发的，要使用 `encode()` 函数以 UTF-8 格式编码 `print` 打印的内容。例如，打印结果的 `title` 时，要使用 `print(result['title']).encode('utf-8')`。如果遇到 `UnicodeEncodeError` 错误，可能就要这么做。Python 3 不受这个问题影响。

★ 提示 ★

前面在填充脚本中已经这么做过了。你可以试着遵守 Google 的 Python 编程风格指南，添加 `main()` 函数，通过它调用一切。此外，还需要下面两行代码。如果你不知道这两行代码的作用，请看这个在线解答，或者翻回 5.7 节。

```
if __name__ == '__main__':
    main()
```

添加这两行代码后，模块便可以独立运行，而且通过 `import` 导入到其他 Python 程序中，其中的代码也不会自动执行。

修改 `read_webhose_key()` 函数的方法有很多，其中最简单的应该是使用 `os.path.isfile()` 函数检查当前目录中有没有 *search.key* 文件。如果没有，那就可以假定密钥的路径是 *../search.key*，即在运行脚本的上一层目录中。为了使用 `isfile()` 函数，要在 *webhose_search.py* 模块的顶部导入 `os` 模块。

13.3 集成到 Rango 应用中

在 *webhose_search.py* 模块中实现搜索功能后，接下来要把它集成到 Rango 应用中。这个过程主要分三步：

1. 创建 *search.html* 模板，扩展自 *base.html* 模板。*search.html* 模板中有个 HTML 表单，收集用户输入的查询词条，以及用于呈现结果的模板代码。
2. 编写一个 Django 视图，调用前面定义的 `run_query()` 函数，渲染 *search.html* 模板。
3. 在 Rango 应用的 *urls.py* 模块中把新视图映射到一个 URL 上。

创建模板

首先创建 *search.html* 模板。把这个模板放在项目 *templates* 目录中的 *rango* 目录里。在文件中写入下述 HTML 标记和 Django 模板代码。

```
{% extends 'rango/base.html' %}
{% load staticfiles %}

{% block title %} Search {% endblock %}

{% block body_block %}
<div>
    <h1>Search with Rango</h1>
    <br/>
    <form class="form-inline" id="user_form"
          method="post" action="{% url 'search' %}">
        {% csrf_token %}
        <div class="form-group">
            <input class="form-control" type="text" size="50"
                   name="query" value="" id="query" />
```

```
        </div>
        <button class="btn btn-primary" type="submit" name="submit"
                value="Search">Search</button>
    </form>

    <div>
        {% if result_list %}
        <h3>Results</h3>
        <!-- 按顺序显示搜索结果 -->
        <div class="list-group">
        {% for result in result_list %}
            <div class="list-group-item">
                <h4 class="list-group-item-heading">
                    <a href="{{ result.link }}">{{ result.title }}</a>
                    </h4>
                    <p class="list-group-item-text">{{ result.summary }}</p>
            </div>
        {% endfor %}
        </div>
        {% endif %}
    </div>
</div>
{% endblock %}
```

上述模板代码主要执行两个任务：

- 在 HTML 表单中显示搜索框和搜索按钮，供用户输入和提交查询词条。
- 渲染模板时，如果传给模板上下文的 `results_list` 对象有内容，那就迭代 `results_list` 对象，渲染里面的结果。上述模板期望每个结果中有 `title`、`link` 和 `summary`，这与前面定义的 `run_query()` 函数是一致的。

在样式方面，我们使用了 Bootstrap 的列表组和行内表单。

下一小结定义的 Django 视图将通过 `results_list` 上下文变量把搜索结果传给模板。`results_list` 一开始为空，因此未搜索到结果时模板不会渲染后半部分。

编写视图

创建好模板后，接下来要添加视图，渲染模板。在 Rango 应用的 *views.py* 模块中添加下述 `search()` 视图。

```
def search(request):
    result_list = []

    if request.method == 'POST':
        query = request.POST['query'].strip()
        if query:
            # 调用前面定义的函数向 Webhose 发起查询，获得结果列表
            result_list = run_query(query)

    return render(request, 'rango/search.html', {'result_list': result_list})
```

学到现在，你应该能理解上述代码了。这里唯一需要注意的地方是对 `run_query()` 函数的调用。为此，我们要导入 *webhose_search.py* 模块。在运行 Django 开发服务之前，请把下述 `import` 语句添加到 Rango 应用的 *views.py* 模块顶部。

```
from rango.webhose_search import run_query
```

添加映射

接下来要把 `search()` 视图映射到一个 URL 上，还要在导航栏中添加一个链接，让用户能找到搜索页面。

- 把 `search()` 视图映射到 URL */rango/search/* 上，并设定 `name='search'` 参数。即，在 Rango 应用的 *urls.py* 模块中添加 `url(r'search/$', views.search, name='search')`。
- 更新 *base.html* 模板中的导航栏，加入搜索页面的链接。不要在模板中硬编码 URL，应该使用 `url` 模板标签。

最后提醒一下，不要忘了在 Django 项目的根目录中创建 *search.key* 文件，写入你的 Webhose API 密钥。

现在可以向 Webhose API 发起查询了，如图 13-3 所示。

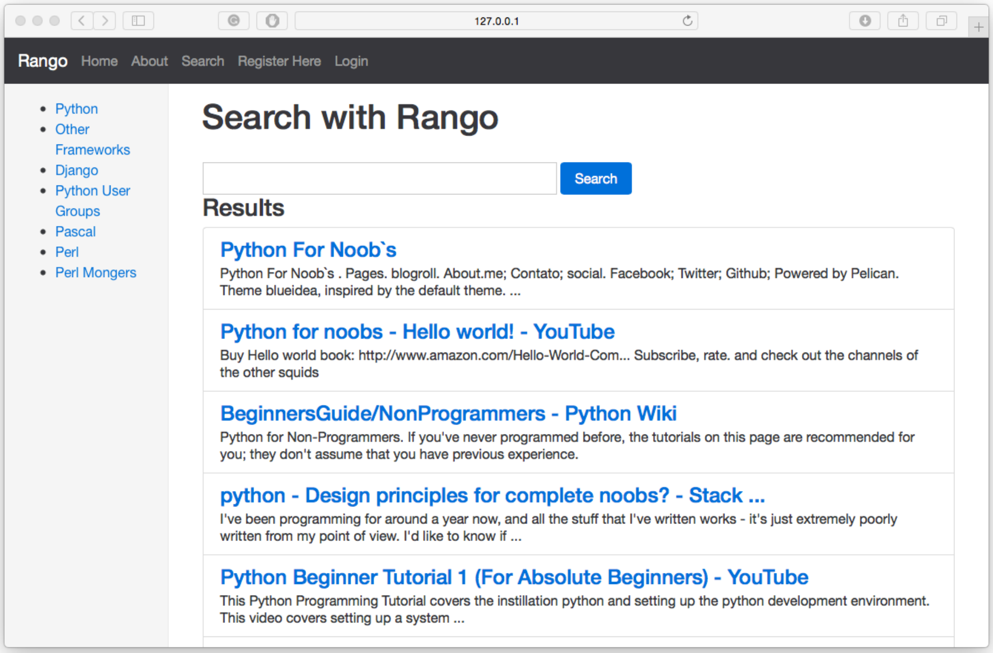


图 13-3：搜索“Python for Noobs”得到的结果

</> 练习

你可能注意到了，显示结果时，搜索词条不见了，这对用户不太友好。修改 `search()` 视图和 *search.html* 模板，在搜索框中显示用户输入的词条。

在视图中，要把 `query` 传入上下文字典。在模板中，要在搜索框中显示查询词条。

14 中期练习

我们一点一点，为 Rango 应用增加了一些功能。之所以每次只增加一个功能，是为了让你熟悉 Django 框架，学习应用程序的各部分是如何组织的。然而，Rango 应用的功能目前还有些松散，而且缺少交互性。本章将给你一些挑战，让你自己动手改进，结合现有的功能和一些新功能，提高应用的用户体验。

为了丰富 Rango 应用的功能，提升交互性，最好再加上下述功能。

- 统计分类和网页的相关数据
  - 分类的查看次数
  - 网页的访问次数
  - 分类的点赞次数（参见第 16 章）
- 在分类页面中集成过滤和搜索功能
  - 删掉独立的搜索页面，让用户在分类页面就能搜索网页
  - 让用户过滤分类，结果显示在侧边栏中（参见第 16 章）
- 为注册用户提供以下服务
  - 假设你已经换用 django-registration-redux，修改注册表单，收集额外的信息（网站和头像）
  - 让用户查看自己的个人资料
  - 让用户编辑自己的个人资料
  - 让用户查看用户列表及其他用户的个人资料

■ 注意 ■

现在不一一实现这些功能，有些在第 16 章完成，有些则留作练习。

着手添加功能之前最好列出要做的事，思考如何完成各个任务。把一项任务分解为一系列小任务能简化实现过程，各个击破。本章将给出部分任务的解决思路。结合目前所学的知识，你应该能自行实现多数功能（用到 Ajax 的除外）。附录 B 将给你一些提示和代码片段，说明如何实现这些功能。当然，如果你真的不知道如何实现，随时可以查看 GitHub 仓库中的代码。

14.1 记录网页的访问次数

目前，Rango 直接提供外部网页的链接。如果想记录每个网页的访问次数，这样做不太合适。为了记录 Rango 应用中各网页的访问次数，要这么做：

- 编写一个视图，命名为 `track_url()`，把它映射到 URL */rango/goto/* 上，并设定 `'name=goto'` 参数。
- `track_url()` 视图从 HTTP GET 请求参数（例如 `/rango/goto/?page_id=1`）中获取 `page_id`。
    - 在视图中，通过 `page_id` 获取 `Page` 对象，增加它的 `views` 字段，然后保存。
    - 使用 Django 的 `redirect` 函数把用户重定向到真正的 URL。
    - 如果 HTTP GET 请求参数中没有 `page_id`，或者找不到对应的 `Page` 对象，重定向到首页。使用 `django.core.urlresolvers` 模块中的 `reverse` 函数获取 URL 字符串，然后重定向。如果使用的是 Django 1.10，可以从 `django.shortcuts` 模块中导入 `reverse` 函数。
    - `redirect` 和 `reverse` 的更多信息参见 Django 文档。
- 更新 *category.html* 模板，使用 */rango/goto/?page_id=XXX*。记得使用 `url` 模板标签，不要硬编码 URL。

```
<a href="{% url 'goto' %}?page_id={{page.id}}">
```

★ GET 参数 ★

如果你不知道如何从 HTTP GET 请求参数中获取 `page_id` 查询字符串，请参照下述代码：

```
    page_id = None
    if request.method == 'GET':
        if 'page_id' in request.GET:
            page_id = request.GET['page_id']
```

一定要先检查请求类型是不是 GET，然后再访问 `request.GET` 字典，获取随请求一起发送的信息。如果其中有 `page_id`，可以使用 `request.GET['page_id']` 获取它的值。

此外也可以不使用查询字符串，而是直接放在 URL 中，例如 */rango/goto/<page_id>/*。这时要使用 URL 模式获取 `page_id`：`r'goto/(?P<page_id>\d+)/$'`。

14.2 在分类页面中搜索

Rango 应用旨在为用户提供对他们有帮助的网页。目前，搜索功能是独立于分类单独存在的。如果能集成到分类页面中的话更好。我们假设用户会先浏览感兴趣的分类，当用户找不到相关网页时，就可以搜索。如果用户找到感兴趣的网页，还可以把网页添加到分类中。现在我们先集中精力解决第一个问题，即在分类页面中添加搜索功能。具体步骤如下：

- 把导航栏中的搜索链接删除，即去掉独立的搜索功能。
- 把 *search.html* 模板中的搜索表单和结果展示代码移到 *category.html* 模板中。
- 修改搜索表单，把 `action` 属性的值改为当前分类页面：

```
<form class="form-inline" id="user_form"
    method="post" action="{% url 'show_category' category.slug %}">
```

- 更新 `show_category()` 视图，处理 HTTP POST 请求。还要把搜索结果传给上下文字典，以便在模板中渲染。
- 此外，限制只让通过身份验证的用户搜索。在 *category.html* 模板中要这样限制访问：

```
{% if user.authenticated %}
    <!-- 在这插入搜索代码 -->
{% endif %}
```

14.3 增加个人资料页面

如果你换用 django-registration-redux 包了，可能会想收集用户的个人资料。此时，不能把用户重

定向到 Rango 应用的首页，而要重定向到一个新的表单，让用户提供头像和个人网站。基本步骤如下：

- 创建 *profile_registration.html* 模板，显示 `UserProfileForm`。
- 定义 `UserProfileFormModelForm` 类，处理这个新表单。
- 编写 `register_profile()` 视图，捕获用户提供的个人信息。
- 把视图映射到 URL *rango/register_profile/* 上。
- 修改 `MyRegistrationView` 类中的 `get_success_url()` 方法，把重定向地址改为 *rango/add_profile/*。

此外，还可以为用户提供查看和编辑个人资料的功能。基本步骤如下：

- 创建 *profile.html* 模板，显示一个表单，添加几个字段，分别显示用户的个人资料（用户名、电子邮件地址、网站和头像）。
- 编写 `profile()` 视图，获取渲染模板所需的数据。
- 把 `profile()` 视图映射到 URL */rango/profile/* 上。
- 修改基模板，在导航栏中添加"Profile"链接，最好与其他用户相关的链接放在一起。这个链接只有已登录的用户才能看到（`{% if user.is_authenticated %}`）。

为了让用户查看其他用户的个人资料，可以创建一个用户列表页面，列出全部用户。点击某个用户后便可查看他的个人资料。不过，必须确保用户只能修改自己的个人资料。

★ 在模板中引用上传的内容 ★

4.3 节讲过如何处理上传的媒体文件，在模板中可以使用 `{{ MEDIA_URL }}` 标签引用 `MEDIA_URL` 地址（在 *settings.py* 模块中设定），例如 `<img src="{{ MEDIA_URL}}cat.jpg">`。

附录 B 将给出一些提示，指导你完成上述功能。

15 jQuery 和 Django

jQuery 是个优秀的 JavaScript 库，免除了很多痛苦。几行 jQuery 代码往往能封装几百行纯 JavaScript 代码。而且，jQuery 提供了一整套处理 HTML 元素的功能。本章将讨论：

- 如何在 Django 应用中使用 jQuery；
- 如何理解 jQuery 代码；
- 提供一些简单的示例。

15.1 在 Django 项目/应用中使用 jQuery

在基模板中添加下述代码：

```
{% load staticfiles %}
<script src="https://cdn.staticfile.org/jquery/3.0.0/jquery.min.js">
<script src="{% static "js/rango-jquery.js" %}"></script>
```

如果你自己下载了 jQuery，保存在静态文件目录中，那就添加下述代码：

```
<script src="{% static "js/jquery.min.js" %}"></script>
<script src="{% static "js/rango-jquery.js" %}"></script>
```

确保你设定了静态文件相关的设置（参见第 4 章）。

在 *static* 目录中新建 *js* 目录，把你下载的 jQuery 脚本文件（*jquery.min.js*）保存在这里。另外，在这个目录中新建一个文件，命名为 *rango-jquery.js*。我们自己编写的 JavaScript 代码保存在这个文件中。打开 *rango-jquery.js*，写入下述 JavaScript 代码：

```
$(document).ready(function() {
    // jQuery 代码写在这里
```

```
});
```

这段代码使用 jQuery 选择文档对象（`$(document)`），然后调用 `ready()` 函数。浏览器准备好文档之后（页面完全加载了），执行 `function() { }` 表示的匿名函数。通常都会等到文档加载完毕才执行 jQuery 函数。否则，执行代码之前，相应的 HTML 元素可能尚未加载。详情参见 jQuery 文档。

★ “选择后执行”模式 ★

> jQuery 代码更偏向函数式编程风格，而纯 JavaScript 代码通常采用过程式编程风格。jQuery 代码遵循的模式是“选择后执行”：先选择一个元素，然后对其执行某种操作。

示例：点击后弹出对话框

下面举个例子，使用标准的 JavaScript 和 jQuery 分别实现相同的功能，让你一窥二者之间的差异。在 *about.html* 模板中添加下述代码：

```
<button  class="btn btn-primary"
    onClick="alert('You clicked the button using JavaScript.');">
    Click Me - I run JavaScript
</button>
```

可以看到，我们把 `alert()` 函数赋值给按钮的 `onClick` 属性。访问关于页面，试试效果。

下面使用 jQuery 实现同样的功能。在 *about.html* 模板中再添加一个按钮：

```
<button  class="btn btn-primary" id="about-btn">
    Click Me - I'm JavaScript on Speed</button>
```

注意，现在按钮上没有任何 JavaScript 代码。相应的代码写在 *rango-jquery.js* 文件中：

```
$(document).ready( function() {
    $("#about-btn").click( function(event) {
        alert("You clicked the button using jQuery!");
    });
});
```

刷新页面，看看效果。理论上点击两个按钮后都会弹出对话框。

这段 jQuery 代码先选择文档对象，在文档准备好之后执行其中的函数，即

`$("#about-btn").click()`。这部分代码从页面中选择 `id` 为 `about-btn` 的元素，然后以编程的方式把 `alert()` 函数赋予 `click` 事件。

一开始你可能会觉得 jQuery 有些麻烦，毕竟做到同一件事，编写的 jQuery 代码多很多。对 `alert()` 这样的简单函数来说，确实如此。但是实现复杂的功能时，使用 jQuery 更方便，因为 JavaScript 代码在单独的文件中。jQuery 在运行时赋予事件句柄，实现了关注点分离，即把 jQuery/JavaScript 代码与 HTML 标记解耦了。

■ 分开是好事 ■

> 关注点分离是一种值得铭记的设计原则。对 Web 应用来说，HTML 负责编写页面的内容，CSS 用于装饰内容的外观，而 JavaScript 负责与内容交互，以及操纵内容和样式。
>
> 各司其职，这样写出的代码更简洁，能减少后期维护的投入。

选择符

jQuery 提供了多种选择元素的方式。前例展示了如何使用 `#` 选择符在 HTML 文档中查找特定 ID 的元素。如果想查找类，使用 `.` 选择符，例如：

```
$(".ouch").click( function(event) {
    alert("You clicked me! ouch!");
});
```

此时，文档中 `class` 属性为 `"ouch"` 的所有元素都被选中，把点击句柄设为 `alert()` 函数。注意，所有被选中的元素都被赋予了同一个函数。

此外，还可以通过标签名选择 HTML 元素：

```
$("p").hover( function() {
    $(this).css('color', 'red');
},
function() {
    $(this).css('color', 'blue');
});
```

把这段 JavaScript 代码添加到 *rango-jquery.js* 文件中，然后在 *about.html* 模板中添加一个段落，`<p>This text is for a JQuery Example</p>`。刷新关于页面，把鼠标悬停在那个段落上看看效果。

这里，我们选择所有 HTML p 元素，并为悬停事件关联了两个函数，一个在鼠标移到元素上时执行，另一个在鼠标从元素上移开时执行。可以看到，这里用到了另一个选择符，即 this。这个选择符表示当前选中的元素。注意，jQuery 的 hover() 函数接受两个函数。

上述代码要添加到 $(document).ready() 函数中。如果把 $(this) 改成 $(p)，情况如何？

悬停是一种鼠标移动事件。其他鼠标移动事件的说明参见 jQuery API 文档。

15.2 示例：操纵 DOM

在前例中，我们使用 hover() 函数为悬停事件指定处理句柄，然后使用 css() 函数修改元素的颜色。css() 函数用于操纵 DOM，除此之外 jQuery 库提供了很多这样的函数。例如，可以使用 addClass() 函数为元素添加类：

```
$("#about-btn").addClass('btn btn-primary');
```

这行代码选择 ID 为 about-btn 的元素，为其添加 btn 和 btn-primary 两个类。

还可以访问某个元素内部的 HTML。把下述 div 元素添加到 *about.html* 模板中：

```
<div id="msg">Hello - I'm here for a JQuery Example too</div>
```

然后在 *rango-jquery.js* 中添加下述 JavaScript 代码：

```
$("#about-btn").click( function(event) {
    msgstr = $("#msg").html()
    msgstr = msgstr + "ooo"
    $("#msg").html(msgstr)
});
```

点击 ID 为 about-btn 的元素后，先获取 ID 为 msg 的元素中的内容，在后面追加 "ooo" 之后，再次调用 html() 函数，传入修改后的内容，替换元素中现有的 HTML。

本章简单介绍了如何在 Django 应用中使用 jQuery，还举例说明了 jQuery 的用法。下一章将使用 jQuery 为 Rango 应用添加 Ajax 功能。

16 使用 jQuery 处理 Ajax 请求

Ajax 是一项综合技术，旨在减少页面加载次数。使用 Ajax，重新加载的不是整个页面，而是页面中的部分内容或数据。如果你没用过 Ajax，或者想在使用之前进一步了解，请阅读 Mozilla 网站中的资料。

为了简化处理 Ajax 请求的过程，我们将使用 jQuery 库。注意，如果你使用了 Twitter Bootstrap，那就已经添加了 jQuery。我们使用的是 jQuery 3。如果不想引用在线版，可以把 jQuery 库下载到本地，保存在项目的 *static/js/* 目录中。

16.1 通过 Ajax 实现的功能

为了提升 Rango 应用的现代感，我们将使用 Ajax 实现一些功能，例如：

- 添加“Like”按钮，让注册用户为分类点赞。
- 添加行内分类建议，让用户在输入的过程中快速找到某个分类。
- 在搜索结果中添加“Add”按钮，方便注册用户把网页添加到分类中。

在 *static/js/* 目录中新建一个文件，命名为 *rango-ajax.js*。在基模板中添加下述内容：

```
<script src="{% static "js/jquery.min.js" %}"></script>
<script src="{% static "js/rango-ajax.js" %}"></script>
```

这里，我们假设你把 jQuery 库下载到本地了。此外还可以直接引用线上版本：

```
<script
    src="https://cdn.staticfile.org/jquery/3.0.0/jquery.min.js">
</script>
```

如果你使用 Bootstrap，滚动到模板文件的底部，你会看到引用 jQuery 库的代码。在 jQuery 库之

后引入 *rango-ajax.js* 文件。

至此，我们引入了 jQuery，也有位置编写客户端 Ajax 代码了。下面开始修改 Rango 应用。

16.2 添加点赞按钮

允许注册用户表示他们“喜欢”某个分类是个不错的功能。我们将实现为分类“点赞”的功能，但不记录用户为哪些分类点了赞。注册用户可以不断刷新页面，多次点击点赞按钮。如果想记录用户点赞了哪些分类，要再定义一个模型，还需要其他辅助设置。这留作练习给你完成。

基本流程

让用户为分类点赞的基本过程如下：

- 打开 *category.html* 模板，添加“Like”按钮，把 ID 设为 `like`；添加一个模板标签，显示点赞次数：`{{% category.likes %}}`；把点赞次数放在 ID 为 `like_count` 的 `div` 元素中：`<div id="like_count">{{ category.likes }}</div>`。`category()` 视图传入了分类对象，因此在模板中可以通过 `{{ category.likes }}` 访问点赞次数。
- 编写一个视图，命名为 `like_category()`，从请求中获取 `category_id` 参数，增加对应分类的点击次数。这个视图不返回 HTML 页面，而是返回对应分类的点赞次数。
- 把 `like_category()` 视图映射到一个 URL 上，例如 *rango/like/*。那么 GET 查询参数为 *rango/like/?category_id=XXX*。
- 在 *rango-ajax.js* 文件中编写 jQuery 代码，执行 Ajax GET 请求。如果请求成功，更新 `#like_count` 元素，并隐藏点赞按钮。

修改分类页面的模板

我们要在模板中添加“Like”按钮，并把按钮的 ID 设为 `like`。此外，还要添加一个 `<div>` 元素，显示点赞次数。打开 *category.html* 模板文件，在 `<h1>{{ category.name }}</h1>` 下方添加下述代码：

```
<div>
    <strong id="like_count">{{ category.likes }}</strong> people like this category
    {% if user.is_authenticated %}
    <button id="likes" data-catid="{{category.id}}"
        class="btn btn-primary  btn-sm" type="button">
```

```
        Like
    </button>
    {% endif %}
</div>
```

编写视图

在 *rango/views.py* 模块中编写一个新视图，命名为 `like_category()`。这个视图从请求中获取 `category_id` 参数，然后增加对应分类的点赞次数。

```
from django.contrib.auth.decorators import login_required

@login_required
def like_category(request):
    cat_id = None
    if request.method == 'GET':
    cat_id = request.GET['category_id']
    likes = 0
    if cat_id:
        cat = Category.objects.get(id=int(cat_id))
        if cat:
            likes = cat.likes + 1
            cat.likes =  likes
            cat.save()
    return HttpResponse(likes)
```

可以看到，我们只允许通过身份验证的用户访问这个视图，因为视图上面有 `@login_required` 装饰器。

注意，这个视图假定 GET 请求参数中有 `category_id` 变量，以此标识要更新的分类。这里还可以记录是哪个用户点的赞，不过为了集中精力讲解 Ajax，一切从简。

别忘了在 *rango/urls.py* 模块中添加 URL 映射。在 `urlpatterns` 列表中添加：

```
url(r'^like/$', views.like_category, name='like_category'),
```

发起 Ajax 请求

现在要在 *rango-ajax.js* 文件中添加一些 jQuery 代码，执行 Ajax GET 请求。

```
$('#likes').click(function(){
    var catid;
    catid = $(this).attr("data-catid");
    $.get('/rango/like/', {category_id: catid}, function(data){
        $('#like_count').html(data);
        $('#likes').hide();
    });
});
```

这段 jQuery/JavaScript 代码为 `#likes` 元素添加一个事件句柄。点击这个按钮后，从按钮元素中提取分类 ID，然后向 */rango/like/* 发起 Ajax GET 请求，并在请求中编码 `category_id`。如果请求成功，使用返回的数据更新 `#like_count` 元素的内容，并把 `#likes` 元素隐藏起来。

这里涉及很多知识，而且 Ajax 对页面结构有一定的要求，容易出错。简单而言，点击按钮后，我们通过 Ajax 请求一个 URL，触发 `like_category()` 视图更新分类，返回点赞次数。Ajax 请求收到响应后，更新页面中的部分内容，即点赞次数，并把 `#likes` 按钮隐藏起来。

16.3 添加行内分类建议

现在，用户要在一个长列表里寻找自己感兴趣的分类，如果能提供一种快速查找分类的方法就好了。为此，我们可以在用户输入的过程中向服务器发送请求，获取一些分类建议，让用户选择。

基本流程

实现行内分类建议的基本步骤如下：

- 定义一个带参数的函数，命名为 `get_category_list(max_results=0,starts_with='')`，如果 `max_results=0`，返回所有以 `starts_with` 开头的分类；否则，最多返回 `max_results` 个分类。
- 编写一个视图，命名为 `suggest_category()`，通过查询字符串获取分类。
    - 假设是 GET 请求，尝试访问 `query` 参数。
    - 如果 `query` 参数不为空，从 `Category` 模型中获取以此开头的前 8 个分类。
    - 通过模板显示分类列表。
    - 不要新建模板了，可以继续使用 *cats.html*，毕竟显示的是同种数据（即分类）。
    - 为了让客户端请求数据，要添加一个 URL 映射，姑且称之为 `suggest` 吧。

在此之后，要更新 *base.html* 模板，加上分类搜索框。然后编写一些 JavaScript/jQuery 代码，获取分类列表，在用户输入的过程中显示。

打开 *base.html* 模板，修改侧边栏，添加一个 `<div>` 元素，ID 为 `cats`，用于显示建议的分类。这个元素的内容由 jQuery/JavaScript 更新。在这个 `<div>` 元素前面添加一个搜索框，供用户输入想搜索的分类。

```
<input  class="input-medium search-query" type="text"
    name="suggestion" value="" id="suggestion" />
```

在模板中添加这些元素之后，编写 jQuery 代码，在用户输入的过程中更新分类列表。

为 `#suggestion` 输入框注册一个按键事件句柄：`$('#suggestion').keyup(function(){ … })`。在键盘回升时通过 `.get()` 函数发起一个 Ajax 请求（`$(this).get(…)`），获取更新的分类列表。如果请求成功，使用 `.html()` 函数，把 `#cats` 元素的内容替换掉（`$('#cats').html(data)`）。

</> 练习

修改填充脚本添加这几个分类：Pascal、Perl、PHP、Prolog、PostScript 和 Programming。这样更能看出分类建议功能的作用。

定义辅助函数

我们可以定义一个辅助方法，使用过滤器查找以指定字符串开头的分类。我们要使用的过滤器是 `istartwith`，即不区分字母的大小写。如果想区分大小写，使用 `startswith`。

```
def get_category_list(max_results=0, starts_with=''):
    cat_list = []
    if starts_with:
        cat_list = Category.objects.filter(name__istartswith=starts_with)

    if max_results > 0:
        if len(cat_list) > max_results:
            cat_list = cat_list[:max_results]
    return cat_list
```

编写视图

编写一个视图，调用 get_category_list() 函数，获取匹配条件的前 8 个分类。

```
def suggest_category(request):
    cat_list = []
    starts_with = ''

    if request.method == 'GET':
        starts_with = request.GET['suggestion']
    cat_list = get_category_list(8, starts_with)

    return render(request, 'rango/cats.html', {'cats': cat_list })
```

注意，这个视图重用了 *rango/cats.html* 模板。

映射 URL

打开 *rango/urls.py* 模块，把下述代码添加到 urlpatterns 列表中：

```
url(r'^suggest/$', views.suggest_category, name='suggest_category'),
```

更新基模板

在基模板的侧边栏中添加下述 HTML 标记：

```
<ul class="nav nav-list flex-column">
    <li class="nav-item">Type to find a category</li>
    <form>
    <li class="nav-item"><input class="search-query form-control" type="text"
        name="suggestion" value="" id="suggestion" />
    </li>
    </form>
</ul>
<hr>
<div id="cats">
</div>
```

我们添加了一个 ID 为 suggestion 的输入框，以及一个 ID 为 cats 的 <div> 元素，用于显示分类建议。这里无需添加按钮，因为我们将为输入框的按键事件注册一个事件句柄，通过 Ajax 获取分

类建议。

把模板中的下面三行代码删除：

```
{% block sidebar_block %}
    {% get_category_list category %}
{% endblock %}
```

通过 Ajax 请求获取分类建议

在 *js/rango-ajax.js* 文件中添加下述 jQuery 代码：

```
$('#suggestion').keyup(function(){
    var query;
    query = $(this).val();
    $.get('/rango/suggest/', {suggestion: query}, function(data){
        $('#cats').html(data);
    });
});
```

我们为 ID 为 `suggestion` 的输入元素注册了一个句柄，在按键回升事件触发时执行。该事件触发时，把输入框中的内容赋值给 `query` 变量，然后向 */rango/suggest/* 发起 Ajax GET 请求。如果请求成功，更新 ID 为 `cats` 的元素，把里面的内容替换为得到的分类建议。

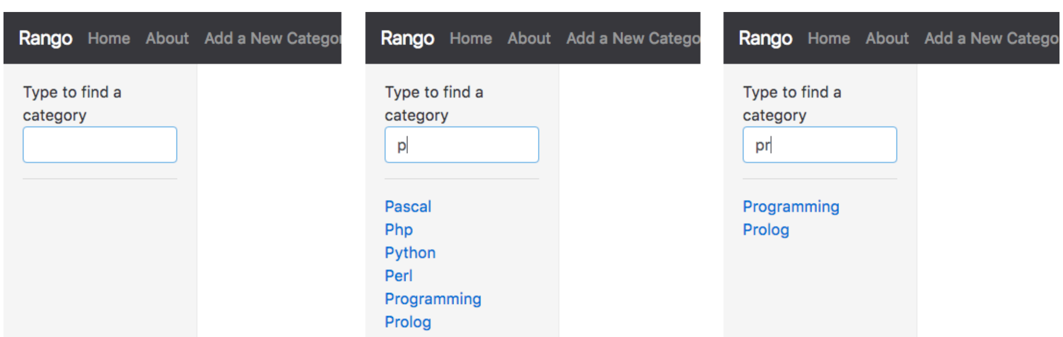


图 16-1：行内分类建议示例。注意建议的分类在输入的过程中是有变化的。

</> 练习

为了方便注册用户把网页添加到分类中，在各搜索结果旁添加一个“Add”按钮。

- 更新 *category.html* 模板，在各搜索结果旁添加“Add”按钮（仅针对通过身份验证的用户）。这个按钮要标有网页的标题和 URL，供 jQuery 使用。
- 把分类页面中的网页列表放在一个 ID 为 `page` 的 `<div>` 元素中，以便添加网页后更新其中的内容。
- 编写一个视图，例如名为 `auto_add_page()`，从 GET 请求参数中获取 `title`、`url` 和 `catid`，把网页添加到相应的分类中。
- 把这个视图映射到一个 URL 上：`url(r'^add/$', views.auto_add_page, name='auto_add_page')`。
- 使用 jQuery 为“Add”按钮添加一个事件句柄，添加网页后隐藏“Add”按钮。

★ 参考解答 ★

在 *category.html* 模板中添加的代码如下。注意，这个按钮要标有一些重要的信息。

```
{% if user.is_authenticated %}
    <button data-catid="{{category.id}}" data-title="{{ result.title }}"
        data-url="{{ result.link }}"
            class="rango-add btn btn-info btn-sm" type="button">Add</button>
{% endif %}
```

然后为类为 `rango-add` 的每个按钮添加 `click` 事件句柄：

```
$('.rango-add').click(function(){
    var catid = $(this).attr("data-catid");
    var url = $(this).attr("data-url");
    var title = $(this).attr("data-title");
    var me = $(this)
    $.get('/rango/add/',
        {category_id: catid, url: url, title: title}, function(data){
            $('#pages').html(data);
            me.hide();
        });
    });
```

视图代码：

```python
@login_required
def auto_add_page(request):
    cat_id = None
    url = None
    title = None
    context_dict = {}
    if request.method == 'GET':
        cat_id = request.GET['category_id']
        url = request.GET['url']
        title = request.GET['title']
        if cat_id:
            category = Category.objects.get(id=int(cat_id))
            p = Page.objects.get_or_create(category=category,
                title=title, url=url)
            pages = Page.objects.filter(category=category).order_by('-views')
            # 把网页列表传给模板上下文
            context_dict['pages'] = pages
    return render(request, 'rango/page_list.html', context_dict)
```

新模板 *page_list.html* 的内容：

```html
{% if pages %}
<ul>
    {% for page in pages %}
    <li><a href="{% url 'goto' %}?page_id={{page.id}}"\>{{ page.title }}</a></li>
    {% endfor %}
</ul>
{% else %}
    <strong>No pages currently in category.</strong>
{% endif %}
```

最后，别忘了添加 URL 映射：

```python
url(r'^add/$', views.auto_add_page, name='auto_add_page'),
```

如果一切顺利，分类页面中的搜索结果如图 16-2 所示。

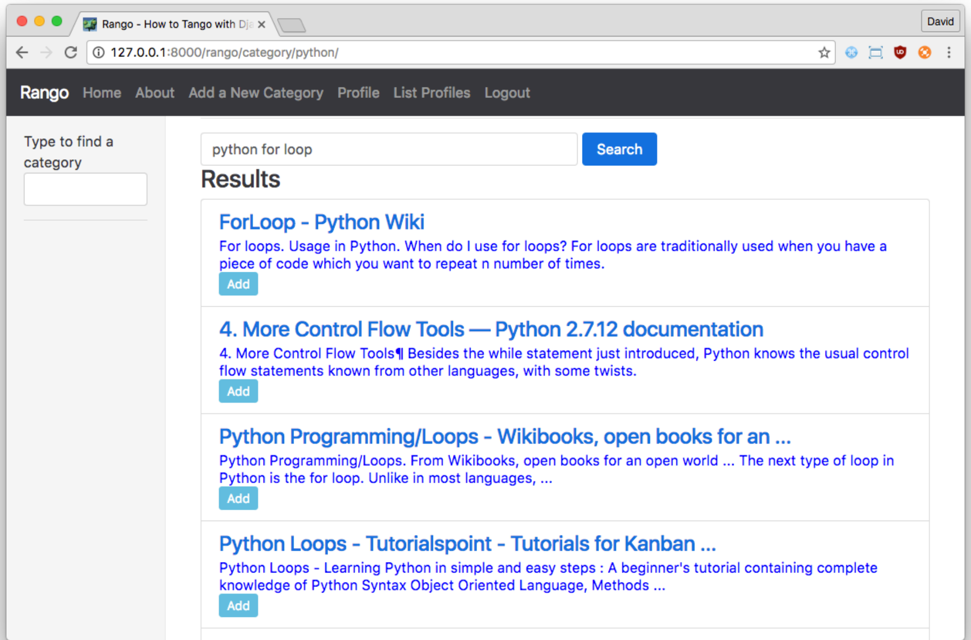


图 16-2：分类页面中搜索结果里的“Add”按钮

17 自动化测试

编写测试是个好习惯，这样能确保软件稳固可靠。当然，有时大部分时间都用在实现功能上，没有多少时间过问软件运行得如何，或者过于自大，确信自己写出的功能一定能用。

编写测试的原因有很多，Django 官方教程列出了几点：

- 测试能节省时间：修改复杂的系统时可能会出现意外失误。
- 测试不仅能找出问题，还能避免问题的发生：测试能指出哪些代码不符合预期。
- 测试能让代码更具魅力：Django 的原始开发者之一 Jacob Kaplan-Moss 说过，“没有测试的代码先天不足”。
- 测试能协助团队协作，防止团队中的成员不小心破坏功能。

根据 Python 指南，编写测试时要遵守一些基本规则。下面是其中几条重要的规则：

- 测试应该针对具体的小功能
- 测试应该有明确的目的
- 测试应该独立
- 在编写代码之前，提交和推送代码之前要运行测试
- 最好创建一个钩子，在推送代码时运行测试
- 测试的名字不怕长，要明确表明意图

★ 说明 ★

目前本章的内容还很基础，而且与 Django 官方教程中的内容差不多，不过增加了一些说明。笔者计划以后再扩充本章的内容。

17.1 运行测试

Django 集成了测试功能。可以执行下述命令测试 Rango 应用：

```
$ python manage.py test rango

Creating test database for alias 'default'...

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
Destroying test database for alias 'default'...
```

这个命令运行 Rango 应用中的所有测试。目前，我们的应用还没什么测试。不信打开 *rango/tests.py* 文件看看，里面只有一个导入语句。每次创建应用，Django 都会自动生成这样一个文件，鼓励你编写测试。

从输出中可以看到，提到了一个名为"default"的数据库。运行测试时，Django 会创建一个临时数据库，供测试填充数据和执行操作。这样，测试时使用的数据库就与线上数据库隔离开了。

17.2 测试模型

下面我们来编写一个测试。对 `Category` 模型的各属性来说，我们希望查看次数的值为零或正数，难道次数能小于零？为了测试这个限制，把下述代码写入 *rango/tests.py* 文件：

```
from django.test import TestCase
from rango.models import Category

class CategoryMethodTests(TestCase):
    def test_ensure_views_are_positive(self):
        """
        ensure_views_are_positive 函数在分类的查看次数
        为零或正数时应该返回 True
        """
        cat = Category(name='test',views=-1, likes=0)
        cat.save()
        self.assertEqual((cat.views >= 0), True)
```

如果你以前没写过测试，首先注意，测试继承自 TestCase 类。其中的方法也有约定，名称以 test_ 开头的方法都是测试用例，里面有一些断言（assertion）。这里，我们使用 assertEqual() 方法判断两个值是否相等。此外还有很多断言方法可用（例如 assertItemsEqual、assertListEqual、assertDictEqual，等等），详情参见文档（Python 2 版，Python 3 版）。Django 的测试机制源自 Python，在此基础上还额外提供了一些断言方法。

下面运行测试：

```
$ python manage.py test rango

Creating test database for alias 'default'...
F
======================================================================
FAIL: test_ensure_views_are_positive (rango.tests.CategoryMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "/Users/leif/Code/tango_with_django_project_19/rango/tests.py",
    line 12, in test_ensure_views_are_positive
    self.assertEqual((cat.views>=0), True)
    AssertionError: False != True

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

可以看到，这个测试失败了。这是因为 Category 模型并没有检查查看次数的值是否小于零。我们确实想保证这个属性的值不为负数，因此要修改模型，加入这一限制。请在 Category 模型的 save() 方法中添加一些代码，检查 views 属性的值。这里只需检查 self.views 的值是否小于零就足够了。

修改模型之后再运行测试，看看能不能通过。如果不能，再修改。

下面再编写一个测试，确保别名的格式正确，即以连字符连接各词，而且全为小写字母。把下述代码添加到 *rango/tests.py* 文件中：

```
def test_slug_line_creation(self):
    """
    slug_line_creation 确保添加分类时创建的别名格式是正确的
    例如 "Random Category String" -> "random-category-string"
```

```
        """
        cat = cat('Random Category String')
        cat.save()
        self.assertEqual(cat.slug, 'random-category-string')
```

现在的代码能让这个测试通过吗？

17.3 测试视图

前一节编写的测试用于确保模型中数据的完整性。Django 还为视图提供了测试机制，使用一个模拟客户端通过 URL 访问 Django 视图。在视图测试中可以访问响应（包括 HTML）和上下文字典。

下面编写一个测试，确认 `Category` 模型为空时，首页会显示"There are no categories present"消息。

```
from django.core.urlresolvers import reverse

class IndexViewTests(TestCase):

    def test_index_view_with_no_categories(self):
        """
        如果没有分类，应该显示一条提示消息
        """
        response = self.client.get(reverse('index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "There are no categories present.")
        self.assertQuerysetEqual(response.context['categories'], [])
```

首先，Django `TestCase` 能访问 `client` 对象，请求就是通过它发送的。这个测试使用 `reverse` 函数查询首页的 URL，然后访问那个页面，得到 `response` 对象。这个测试检查了几件事：首页能不能成功加载，响应 HTML 中是否包含"There are no categories present."这句话，以及上下文字典中的分类列表是否为空。别忘了，运行测试时会创建一个新数据库，但是默认没有填充任何数据。

下面添加几个分类，然后再测试视图。首先定义一个辅助方法：

```
from rango.models import Category

def add_cat(name, views, likes):
    c = Category.objects.get_or_create(name=name)[0]
```

```
        c.views = views
        c.likes = likes
        c.save()
        return c
```

然后在 IndexViewTests 类中再添加一个测试方法：

```
    def test_index_view_with_categories(self):
        """
        确保首页显示了分类
        """

        add_cat('test',1,1)
        add_cat('temp',1,1)
        add_cat('tmp',1,1)
        add_cat('tmp test temp',1,1)

        response = self.client.get(reverse('index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "tmp test temp")

        num_cats =len(response.context['categories'])
        self.assertEqual(num_cats , 4)
```

这个测试先向数据库中填充 4 个分类，然后检查页面中有没有“tmp test temp”文本，以及分类数量是不是 4 个。注意，这里做了三项检查，但是同属一个测试。

17.4 测试渲染的页面

使用 Django 的测试客户端和（或）Selenium 还可以启动应用，以编程的方式测试 HTML 页面中的 DOM 元素。这里不再举例说明。

17.5 测试覆盖度

覆盖度衡量有多少代码已被测试，还有多少尚未得到测试。可以安装 coverage 包（pip install coverage），让它为我们统计覆盖度。安装好之后，执行下述命令：

```
$ coverage run --source='.' manage.py test rango
```

这个命令检查 Rango 应用的所有测试，汇总覆盖度数据。若想查看覆盖度报告，执行下述命令：

```
$ coverage report
```

```
Name                                        Stmts   Miss  Cover
---------------------------------------------------------------
manage                                          6      0   100%
populate                                       33     33     0%
rango/__init__                                  0      0   100%
rango/admin                                     7      0   100%
rango/forms                                    35     35     0%
rango/migrations/0001_initial                   5      0   100%
rango/migrations/0002_auto_20141015_1024        5      0   100%
rango/migrations/0003_category_slug             5      0   100%
rango/migrations/0004_auto_20141015_1046        5      0   100%
rango/migrations/0005_userprofile               6      0   100%
rango/migrations/__init__                       0      0   100%
rango/models                                   28      3    89%
rango/tests                                    12      0   100%
rango/urls                                     12     12     0%
rango/views                                   110    110     0%
tango_with_django_project/__init__              0      0   100%
tango_with_django_project/settings             28      0   100%
tango_with_django_project/urls                  9      9     0%
tango_with_django_project/wsgi                  4      4     0%
---------------------------------------------------------------
TOTAL                                         310    206    34%
```

从上述报告可以看出，很多重要的代码还没有测试，例如 *rango/views* 中的视图。coverage 包功能丰富，能协助你编写更全面的测试。

</> 练习

假设我们想扩展 Page 模型，添加两个字段：last_visit 和 first_visit。这两个字段的类型都是 timedate。

- 更新 Page 模型，添加这两个字段。

- 更新添加网页功能和转向功能。
- 编写测试，确保最后访问和首次访问时间不是将来的时间。
- 编写测试，确保最后访问时间与首次访问时间相同，或者在首次访问时间之后。
- 阅读 Django 官方教程第五部分，进一步学习测试。
- 阅读 Harry Percival 写的《Test-Driven Development with Python》。[1]

1. 《Test-Driven Development with Python》一书的中文版由本书译者翻译，由人民邮电出版社出版。

18 部署 Django 项目

本章带领你一步一步部署 Django 应用。我们将把应用部署到 PythonAnywhere 上，这是一个在线 IDE 和托管服务，在浏览器中就能访问服务器端的 Python 和 Bash 命令行界面。因此，在你的电脑中就能通过终端操作 PythonAnywhere 的服务器。PythonAnywhere 的免费账户提供了一定量的存储空间和 CPU 时间，足够运行 Django 应用。

18.1 注册 PythonAnywhere 账户

首先，注册一个 PythonAnywhere 入门账户。如果你的应用发展形式好，深受欢迎，随时可以升级套餐，获得更多的存储空间和 CPU 时间，以及其他好处，例如自定义域名和 SSH。

注册账户后，PythonAnywhere 会为你分配一个网址，格式为 *http://<username>.pythonanywhere.com*，其中 `<username>` 是你的 PythonAnywhere 用户名。你的应用就通过这个 URL 访问。

18.2 PythonAnywhere 的 Web 界面

PythonAnywhere 的 Web 界面有个控制面板，用于管理你的应用，如图 18-1 所示。

- ❑ Consoles 组件：创建并与 Python 和 Bash 控制台交互。
- ❑ Files 组件：上传和管理文件。
- ❑ Web Apps 组件：配置托管的 Web 应用。

此外还有 Notebooks 等组件，但是我们现在用不到。本章主要使用 Consoles 和 Web Apps 组件。如果想了解其他组件的用法，请阅读 PythonAnywhere 维基中的详细说明。

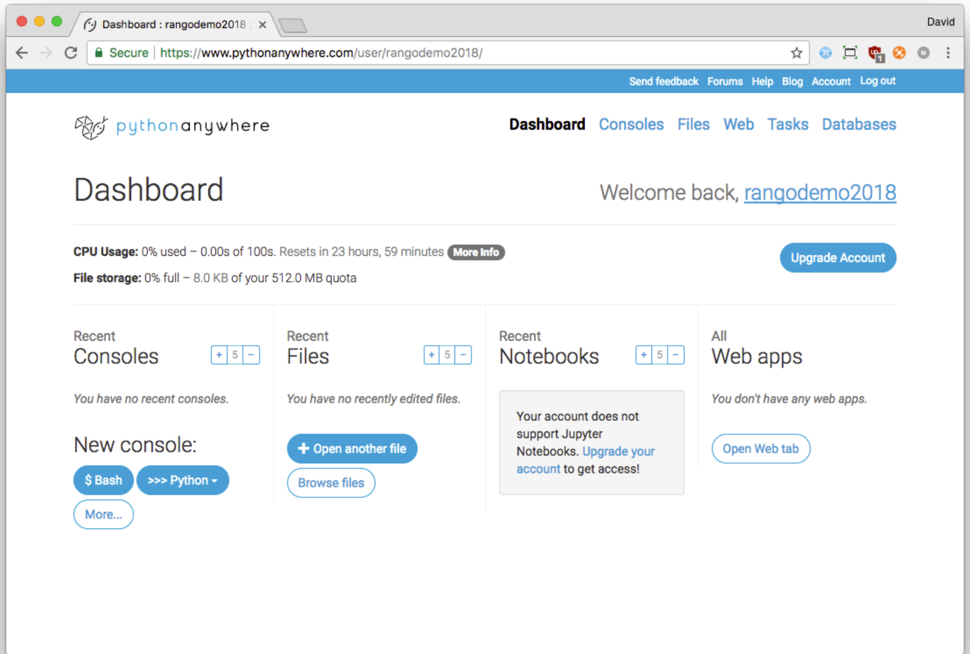


图 18-1：PythonAnywhere 的控制面板，显示着可用的主要组件

18.3 搭建虚拟环境

PythonAnywhere 的默认 Bash 环境自带 Python 2.7.6 和一些包（包括 Django 1.3.7 和 Django-Registration 0.8）。这与我们所需的环境不同，因此我们要创建一个虚拟环境。

首先，打开 Bash 控制台。在 PythonAnywhere 的控制面板中点击 Consoles 组件下方的“$ Bash”按钮。此时会出现一个黑色界面，初始化终端。终端初始化完成后（看到时间），输入下述命令：

```
$ mkvirtualenv --python=<python-version> rango
```

如果你在阅读本书的过程中使用的是 Python 3.x，把 `<python-version>` 替换为 `python3.4`、`python3.5` 或 `python3.6`（在你的电脑中执行 `python --version` 命令，确认版本号）。如果你使用的是 Python 2.7.x，把 `<python-version>` 替换为 `python2.7`。上述命令使用你指定的 Python 版本创建一个虚拟环境，名为 `rango`。下面是笔者创建 Python 3.6 虚拟环境得到的输出：

```
11:45 ~ $ mkvirtualenv --python=python3.6 rango
Running virtualenv with interpreter /usr/bin/python3.6
New python executable in /home/rangodemo2018/.virtualenvs/rango/bin/python3.6
Also creating executable in /home/rangodemo2018/.virtualenvs/rango/bin/python
```

```
Installing setuptools, pip, wheel...done.
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../predeactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../postdeactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../preactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../postactivate
virtualenvwrapper creating /home/rangodemo2018/.virtualenvs/.../get_env_details
```

注意，这段输出中的 PythonAnywhere 用户名是 `rangodemo2018`。你看到的将是你自己的用户名。创建虚拟环境要花一点时间，创建完毕后你会看到一个稍微不同的提示符：

```
(rango) 11:45 ~ $
```

注意，与之前相比，现在多了 `(rango)`。这表示 `rango` 虚拟环境已激活，所有包都将安装在这个虚拟环境中，不会影响系统全局。现在执行 `ls -la` 命令，你会看到有个 *.virtualenvs* 目录。虚拟环境和相关的包都存储在这里。为了确认一切正常，执行 `which pip` 命令。这个命令打印当前激活的 `pip` 二进制文件的位置，我们希望它在 `rango` 虚拟环境中。

```
/home/<username>/.virtualenvs/test/bin/pip
```

若想查看安装了哪些包，执行 `pip list` 命令。接下来我们要定制虚拟环境，安装 Rango 应用所需的包。执行下述命令：

```
$ pip install -U django==1.9.10
$ pip install pillow
$ pip install django-registration-redux
$ pip install django-bootstrap-toolkit
```

切记把 Django 的版本替换为你在开发过程中使用的版本。如若不然，很有可能遇到各种奇怪的问题，让你百思不解。你也可以在自己的电脑中执行 `pip freeze > requirements.txt` 命令，保存当前开发环境，然后在 PythonAnywhere 中执行 `pip install -r requirements.txt` 命令，一次安装全部包。

★ 静候下载…… ★

安装包的过程可能要花些时间，趁机你可以放松一下、给朋友打个电话或者发推评价一下本书。

安装好之后，执行 `which django-admin.py` 命令检查 Django 有没有安装。正常情况下应该看到类似下面的输出：

```
/home/<username>/.virtualenvs/rango/bin/django-admin.py
```

★ PythonAnywhere 上的虚拟环境 ★

PythonAnywhere 也提供了搭建虚拟环境的说明，详情参见维基。

切换虚拟环境

在虚拟环境之间切换十分容易。PythonAnywhere 能为你代劳。下面简要说明如何切换虚拟环境。

若想进入某个虚拟环境，在终端执行 `workon` 命令。例如，下述命令进入 `rango` 环境：

```
16:48 ~ $ workon rango
```

请把 `rango` 换成你想使用的虚拟环境名称。执行这个命令之后，提示符会改变，指明你在一个虚拟环境中。例如下方的 `(rango)`。

```
(rango) 16:49 ~ $
```

如果想离开虚拟环境，执行 `deactivate` 命令。离开后，提示符中的 `(rango)` 会消失不见，如下所示：

```
(rango) 16:49 ~ $ deactivate
16:51 ~ $
```

克隆 Git 仓库

搭建好虚拟环境之后，克隆 Git 仓库，获取项目文件。执行下述命令克隆仓库：

```
$ git clone https://<USERNAME>:<PASSWORD>@github.com/<OWNER>/<REPO_NAME>.git
```

请把 `<USERNAME>` 替换为你的 GitHub 用户名，把 `<PASSWORD>` 替换为你的 GitHub 账户密码，把 `<OWNER>` 替换为仓库拥有者的用户名，把 `<REPO_NAME>` 替换为仓库的名称。

设置数据库

源码克隆好之后，接下来要准备数据库。我们将使用本书前面创建的 *populate_rango.py* 模块填充数据库，为此，确保你在 `rango` 虚拟环境中（即提示符前有 `(rango)`）。如果不在这个虚拟环境

中，执行 `workon rango` 命令。从家目录进入 *tango_with_django_19* 目录，再进入 *code* 目录。*tango_with_django_19* 目录应该跟你的 Git 仓库同名，如果你的仓库名为 *twd*，那这个目录的名称就是 *twd*。如果你的仓库结构与这里不同，你要进入 *manage.py* 脚本所在的目录。然后执行下述命令：

```
(rango) 16:55 ~/tango_with_django $ python manage.py makemigrations rango
(rango) 16:55 ~/tango_with_django $ python manage.py migrate
(rango) 16:56 ~/tango_with_django $ python populate_rango.py
(rango) 16:57 ~/tango_with_django $ python manage.py createsuperuser
```

第一个命令为 Rango 应用创建迁移，`migrate` 命令创建 SQLite3 数据库。创建好数据库之后，填充数据，再创建一个超级用户。

18.4 设置 Web 应用

接下来要配置 PythonAnywhere 的 Nginx Web 服务器，伺服我们的应用。在 PythonAnywhere 的控制面板中点击“Open Web tab”按钮。在打开的页面中，点击“Add a new web app”。

此时会弹出一个对话框。按照页面中的说明填写信息，随后点击“manual configuration”（手动配置），然后关闭向导。注意，选择的 Python 版本要与创建虚拟环境时指定的一致。

在 Web 浏览器中打开一个新标签页或窗口，访问 PythonAnywhere 为你分配的二级域名，*http://<username>.pythonanywhere.com*。现在你应该看到默认的“Hello, World!”页面，如图 18-2 所示。这是因为现在伺服网页的是 WSGI 脚本，而不是你的 Django 应用。下面我们就要改过来。

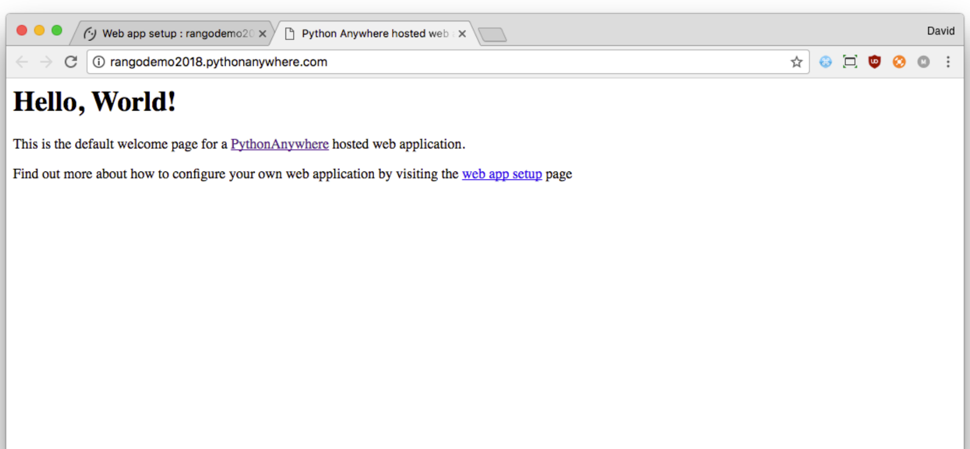


图 18-2：PythonAnywhere 默认的“Hello, World!”页面

配置虚拟环境

在 PythonAnywhere 界面中的“Web”标签页，向下滚动，直到看到“Virtualenv”标题。

输入虚拟环境的路径。假设你的虚拟环境名为 `rango`，那么输入的路径是：

```
/home/<username>/.virtualenvs/rango
```

你可以打开控制台，确认一下。

然后找到“Code”部分，设定 Web 应用源码的路径：

```
/home/<username>/<path-to>/tango_with_django_project/
```

注意，这个路径应该指向 *manage.py* 文件所在的目录。假如你从名为 `tango_with_django_19` 的仓库中克隆源码，放在服务器的家目录中，那么这个路径应该设为：

```
/home/<username>/tango_with_django_19/code/tango_with_django_project/
```

配置 WSGI 脚本

Web Server Gateway Interface（WSGI，Web 服务器网关接口）为 Web 服务器和 Web 应用提供通用的接口。PythonAnywhere 使用 WSGI 建立服务器与应用之间的连接，把入站请求映射到为你的 Web 应用分配的二级域名。

下面配置 WSGI 脚本。在 PythonAnywhere 的界面中打开“Web”标签页，在“Code”标题下有个链接，指向 WSGI 文件，路径为 */var/www/<username>_pythonanywhere_com_wsgi.py*。

PythonAnywhere 为我们提供了一个示例 WSGI 文件，为多个不同的使用场景提供了配置。这里，我们要配置 Django 那部分。点击链接，打开编辑器。下面是针对 Rango 应用的配置。

```
import os
import sys

# 把项目所在的目录添加到 PYTHONPATH 中
path = '/home/<username>/<path-to>/tango_with_django_project/'
if path not in sys.path:
    sys.path.append(path)

# 进入项目所在的目录
os.chdir(path)
```

```
# 告诉 Django，settings.py 模块在哪里
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
                      'tango_with_django_project.settings')

# 导入 Django 项目的配置
import django
django.setup()

# 导入 Django WSGI，处理请求
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

记得把 `<username>` 替换为你的 PythonAnywhere 用户名，并根据实际情况修改上述配置。WSGI 配置脚本中的其他代码都应删除，以防冲突。

这个脚本把项目所在的目录添加到 `PYTHONPATH` 中，这样 Python 才能访问到项目中的模块。如果有其他路径要添加到 `PYTHONPATH` 中，可以继续添加。随后指定项目的 *settings.py* 模块所在的位置。最后，导入 Django 的 WSGI 处理程序，启动应用。

修改好 WSGI 配置之后，点击右上角的"Save"按钮。回到"Web"标签页，点击页面顶部的"Reload"按钮（那个绿色的大按钮）。应用重新加载之后，再次访问 PythonAnywhere 为你分配的二级域名。如果一切顺利，你应该看到应用正常运行起来了。否则，仔细检查 WSGI 配置脚本和各个路径。如果遇到 `DisallowedHost` 异常，参照下一小节的步骤解决。

★ Bad Gateway 错误 ★

在测试的过程中，笔者发现有时会出现"HTTP 502 - Bad Gateway"错误。遇到这个错误，请重新加载应用，然后等一段时间。如果问题依旧，再重新加载试试。如果这个问题始终存在，查看日志文件，看看有没有报错，然后联系 PythonAnywhere 的客服。

接受你的主机名

Django 新版引入了一项安全措施，即接受的主机。只允许 Web 服务器伺服特定的域名，能降低应用被 HTTP Host 首部攻击的影响。首次运行应用时，可能会遇到 `DisallowedHost` 异常，阻碍应用加载。

这个问题容易解决，只需修改项目的 *settings.py* 模块。首先，记下 PythonAnywhere 为你分配的二

级域名。入门账户的域名格式为 *http://<username>.pythonanywhere.com*，其中 `<username>` 是你的 PythonAnywhere 用户名。最好在本地（你的电脑中）编辑这个文件，然后把改动提交到 Git 仓库，推送到远程仓库之后，再拉取到 PythonAnywhere 的服务器。当然，也可以在 PythonAnywhere 的 Web 界面中编辑这个文件，或者在终端里使用文本编辑器（如 nano 或 vi）编辑。

打开项目的 *settings.py* 模块，找到 `ALLOWED_HOSTS` 列表（文件顶部附近）。这个列表默认是空的，把 PythonAnywhere 为你分配的二级域名添加进去：

```
ALLOWED_HOSTS = ['http://<username>.pythonanywhere.com']
```

如果你是在本地电脑中编辑的，现在要执行几个命令，把改动推送到 Git 仓库中：`git add settings.py`、`git commit` 和 `git push`。然后执行 `git pull` 命令，把改动拉取到 PythonAnywhere 的服务器中。如果你是在 PythonAnywhere 中编辑的，直接保存文件即可。

接下来，点击 PythonAnywhere 界面中的“Reload”按钮，重新加载应用。现在访问 PythonAnywhere 为你分配的二级域名，应该看到应用能正常运行起来了。但是，静态文件还不能用。

设定静态文件路径

部署工作就快结束了。目前还有一个问题要解决：为应用设定一些路径。比如，我们要让 PythonAnywhere 的服务器能找到静态文件。

在 PythonAnywhere 的控制面板中点击应用的 URL，在打开的页面中找到“Static files”标题。我们要填写正确的 URL 和文件系统路径，让 PythonAnywhere 的服务器找到并伺服静态文件。

首先，设定 Django 管理后台的静态文件位置。点击“Enter URL”链接，输入 */static/admin/*，点击勾号确认。然后点击“Enter path”链接，输入下述文件系统路径：

```
/home/<username>/.virtualenvs/rango/lib/<python-version>/site-packages/django/
    contrib/admin/static/admin
```

同样，要把 `<username>` 换成你的 PythonAnywhere 用户名。此外，根据你使用的 Python 版本，把 `<python-version>` 替换为 `python2.7`、`python3.6` 等。如果你的虚拟环境没有命名为 `rango`，还要把它替换为你自己的值。输入完成后按回车键，或者点击勾号。

以同样的步骤添加 URL */static/* 和文件系统路径 */home/<username>/<path-to>/tango_with_django_project/static*，设定 Web 应用的 `static` 目录。

设定好之后，点击页面顶部的“Reload”按钮，重新加载应用。注意，可能会出现 HTTP 502 - Bad Gateway 错误。重新加载后，你的 Web 应用应该能正确显示图像等静态文件了。

搜索 API 密钥

把你的搜索 API 密钥保存在 *search.key* 文件中，供 Rango 的搜索功能使用。

关闭调试模式

应用运行起来之后，最好告诉 Django，现在你的应用贮存在生产服务器中。为此，打开项目的 *settings.py* 文件，把 `DEBUG = True` 改成 `DEBUG = False`。这样修改之后，Django 的调试模式关闭了，包含敏感信息的错误消息也不会显示了。尽管如此，你还是能看到出错时的堆栈跟踪（参见下一节）。

18.5 日志文件

把 Web 应用部署到线上环境又增加了一层复杂度。换了环境，可能出现新的问题，有些还让人摸不着头脑。遇到错误时，可以在 PythonAnywhere 创建的三个日志文件中寻找线索。

这些日志文件可以在 PythonAnywhere 界面中的“Web”标签页里查看，也可以在 Bash 控制台中直接查看 */var/log/* 目录里的文件。

PythonAnywhere 创建的三个日志文件如下：

- *access.log*：记录对应用的请求信息
- *error.log*：记录 Web 应用产生的错误消息
- *server.log*：记录运行应用的 Unix 进程的详细信息

注意，各日志文件名的前面有 PythonAnywhere 为你分配的二级域名。例如 *access.log* 文件的全名为 *<username>.pythonanywhere.com.access.log*。

</> 练习

祝贺你，你成功部署了 Rango 应用！

- 把你的应用网址通过 Twitter 布告天下，并 @tangowithdjango。
- 通过 Twitter 或电子邮件告诉我们你对本书的看法。

1.9 结语

本书带领你开发了一个完整的 Web 应用，从提出要求到最终部署，整个过程一步不落。在这个过程中，本书展示了如何为 Web 应用编写模型、视图和模板。此外还用到了一些工具包和服务，例如 Bootstrap、jQuery 和 PythonAnywhere。然而，前面的路还很长。我们只是勾勒出了 Rango 应用的整体结构，还有很多地方可以改进。而改进细节的时候往往要花更多的时间和精力。希望本书能为你打下坚实的基础，让你能大步向前，不断进取。

如果你发现本书内容有问题，欢迎通过 GitHub 反馈给我们。

感谢你阅读本书！

A 设置系统

本篇附录简要说明开发 Django 应用所需的几个组件。

★ 选择一个 Python 版本 ★

Django 既支持 Python 2.7.x，也支持 Python 3。虽然这只是同一门编程语言的两个版本，但是二者之间有本质上的区别。本篇附录假设你使用 Python 2.7.5。请根据自己的需要安装其他版本。

A.1 安装 Python

你的电脑中可能已经预装了 Python。如果你使用的是 Linux 发行版或 macOS，肯定是安装了的。因为这些操作系统的部分功能就是使用 Python 实现的，所以必须要有 Python 解释器。不过，几乎所有现代的操作系统预装的 Python 版本都比本书所要求的版本低。

Python 有很多安装方式，而且大都繁杂。下面说明最常用的安装方式，再给出一些链接，供你参考。

◆ 不要删除默认安装的 Python ◆

本节说明如何在不动预装 Python 的情况下安装 Python 2.7.5。千万不要把操作系统默认安装的 Python 删掉，或者替换为较新的版本。否则，操作系统的部分功能可能失效。

macOS

在 macOS 系统中安装 Python 2.7.5 最简单的方式要数官方提供的安装程序。Python 安装程序的下载地址为 http://www.python.org/getit/releases/2.7.5/。

> **■ 注意选择版本 ■**
>
> 一定要下载针对相应 macOS 版本的 *.dmg* 文件。

1. 双击下载得到的 *.dmg* 文件，弹出一个 Finder 窗口，挂载安装程序。
2. 双击 *Python.mpkg*，打开 Python 安装程序。
3. 不断点击按钮，直到开始安装。安装之前，系统可能会要求你输入密码。
4. 安装完毕后关闭安装程序，弹出挂载的磁盘。删除 *.dmg* 文件。

这样就安装好了较新的 Python 版本，可以开始开发 Django 应用了。简单吧？如果你想使用 Python 3，也可以这样安装。

Linux 发行版

在 Linux 发行版上安装 Python 的方式多种多样，而且不同的发行版也有所区别。例如，在 Fedora 上安装 Python 的方式就与在 Ubuntu 上安装的方式不同。

但也不要灰心。有个工具能帮我们解决这个这个难题——pythonbrew。使用这个工具能轻易安装和管理不同的 Python 版本，而且对操作系统默认的 Python 没有影响。

根据 pythonbrew 项目的说明和 Stack Overflow 中的这个问答，在 Linux 发行版中安装 Python 2.7.5 的步骤如下。

1. 打开终端。
2. 执行命令 `curl -kL http://xrl.us/pythonbrewinstall | bash`，下载 pythonbrew 安装脚本，在终端里执行。pythonbrew 安装在 *~/.pythonbrew* 目录中（~ 表示家目录）。
3. 在文本编辑器（例如 gedit、nano、vi 或 emacs）中打开 *~/.bashrc* 文件，把这一行添加到末尾：`[[ -s $HOME/.pythonbrew/etc/bashrc ]] && source $HOME/.pythonbrew/etc/bashrc`。
4. 保存 *~/.bashrc* 文件，关闭当前终端窗口，重新打开一个，让改动生效。

5. 执行命令 `pythonbrew install 2.7.5`，安装 Python 2.7.5。
6. 执行 `pythonbrew switch 2.7.5` 命令，切换到 Python 2.7.5。

★ 隐藏的目录和文件 ★

> 名称以点号开头的目录和文件相当于 Windows 系统中的隐藏文件。点文件通常在目录浏览工具中不显示，经常用作配置文件。执行 `ls` 命令时加上 `-a` 选项能看到隐藏文件，即 `ls -a`。

Windows

微软 Windows 系统没有预装 Python。因此我们无需担心会对现有的版本有什么影响，直接安装就好了。可以从 Python 官网下载 64 位或 32 位的安装程序。如果不知道该下载哪个版本，按照微软网站中的说明查看自己的电脑是 32 位还是 64 位。

1. 安装程序下载好之后，双击打开。
2. 按照界面上的提示安装 Python。
3. 安装完毕后关闭安装程序，把安装程序删除。

默认情况下，Python 2.7.5 安装在 *C:\Python27* 文件夹中。建议不要改动这个路径。

安装完成后，打开命令提示符，输入 `python` 命令。如果看到 Python 提示符，说明安装是成功的。然而，安装程序有时无法正确设定 `Path` 环境变量，导致找不到 `python` 命令。在 Windows 7 中可以这样解决：

1. 点击“开始”按钮，在“计算机”上点击鼠标右键，在弹出的菜单中选择“属性”。
2. 点击“高级系统设置”链接。
3. 点击“环境变量”按钮。
4. 在“系统变量”列表中找到 `Path`，选中，然后点击“编辑”按钮。
5. 在末尾添加 `;C:\python27;C:\python27\scripts`。别漏了分号，也别添加任何空格。
6. 点击各对话框中的“确定”按钮，保存改动。
7. 关闭命令提示符，重新打开，再执行 `python` 命令试试。

这样 Python 就能正常使用了。在 Windows 10 中的步骤稍有不同。

A.2 设置 PYTHONPATH

安装好 Python 之后，我们要检查安装是否成功。为此要检查有没有正确设置 PYTHONPATH 环境变量。PYTHONPATH 变量告诉 Python 解释器增强 Python 功能的包和模块在什么位置。如未正确设置 PYTHONPATH，我们便无法安装和使用 Django。

首先，确认 PYTHONPATH 变量是否存在。有的安装方法可能会设定这个变量，有的则不会。在基于 Unix 的操作系统中执行下述命令：

```
$ echo $PYTHONPATH
```

在 Windows 系统中执行执行：

```
$ echo %PYTHONPATH%
```

如果一切正常，应该看到类似下面的输出。在 Windows 中显然会看到 Windows 风格的路径，很有可能是 C 盘下的路径。

```
/opt/local/Library/Frameworks/Python.framework/
    Versions/2.7/lib/python2.7/site-packages:
```

这是 Python 安装目录下的 *site-packages* 目录，额外的 Python 包和模块都保存在这里。如果看到类似的路径，请跳到 A.3 节。在 Windows 系统中，*site-packages* 目录在 Python 安装目录下的 *Lib* 目录里。如果 Python 安装在 *C:\Python27*，那么 *site-packages* 目录的路径是 *C:\Python27\Lib\site-packages*。

基于 Unix 的操作系统要经过一番曲折才能找出 *site-packages* 目录的路径。打开 Python解释器，执行下述命令：

```
$ python

Python 2.7.5 (v2.7.5:ab05e7dd2788, May 13 2013, 13:18:45)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import site
>>> print(site.getsitepackages()[0])

'/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages'

>>> quit()
```

`site.getsitepackages()` 返回一组存储额外 Python 包和模块的路径，第一个通常就是 *site-packages* 目录的路径。如果报错 `getsitepackages()` 不在 `site` 模块中，确保你运行的是正确的 Python 版本。2.7.5 版应该有这个函数，之前的版本则没有。

找到 *site-packages* 目录的路径后，将其添加到配置中。在基于 Unix 的操作系统中，再次编辑 *.bashrc* 文件，把下述代码添加到文件末尾：

```
export PYTHONPATH=$PYTHONPATH:<PATH_TO_SITE-PACKAGES>
```

把 `<PATH_TO_SITE-PACKAGES>` 替换为 *site-packages* 目录的具体路径。保存文件后，关闭当前终端窗口，重新打开一个。

在 Windows 系统中，参照前文，打开环境变量设置对话框。添加 `PYTHONPATH` 变量，把值设为 *site-packages* 目录的路径，通常是 *C:\Python27\Lib\site-packages*。

A.3 使用 setuptools 和 pip

搭建开发环境是任何项目的基础工作。虽然可以单独安装 Python 包，但是这会导致诸多问题，为以后带来许多麻烦。例如，你怎么把自己的环境分享给其他开发者呢？如何在新设备中搭建相同的环境呢？如何升级到包的最新版呢？使用包管理器搭建环境能免去很多麻烦事。

本书使用的包管理器是 pip，这是对 Python 包管理器 setuptools 的包装，对用户友好。因为 pip 依赖 setuptools，因此这两个工具都要安装。

首先，从 Python 包官方网站下载 setuptools。可以下载 *.tar.gz* 压缩文件。使用你喜欢的解压工具，把里面的文件提取出来。解压出来的目录通常是 *setuptools-1.1.6*，其中 `1.1.6` 是 setuptools 的版本号。打开终端，进入这个目录，运行 *ez_setup.py* 脚本：

```
$ cd setuptools-1.1.6
$ sudo python ez_setup.py
```

这里我们使用 `sudo` 运行脚本，进行系统全局安装。如果安装成功，会看到类似下面的输出：

```
Finished processing dependencies for setuptools==1.1.6
```

当然，`1.1.6` 会显示为你安装的具体版本号。如果能看到这行输出，接下来就可以安装 pip 了。在终端里执行下述命令：

```
$ sudo easy_install pip
```

这个命令下载并安装 pip，同样也是系统全局安装。如果看到下述输出，说明安装是成功的：

```
Finished processing dependencies for pip
```

随后便可以在终端执行 `pip` 命令。既然已经成功安装，你不会看到无法识别命令的提醒，而是会看到 pip 支持的一系列命令和选项。

> **Windows 中没有 sudo**
>
> 在 Windows 中的安装过程基本差不多，但是无需输入 `sudo`。

A.4 虚拟环境

开发环境就要搭建好了。目前所做的一切足够我们开始着手开发了，但是还有一些不足。如果另一个 Python 应用需要使用不同的版本该怎么办呢？或者，如果你想切换到 Django 的新版，但是想继续维护 Django 1.7 项目又该怎么办呢？

这些问题的解决方法是使用虚拟环境。虚拟环境能让多个 Python 版本及相同包的不同版本和谐相处。如今的开发者都选择使用这种方式搭建 Python 开发环境。只要有 pip，一切都好办。为了使用虚拟环境，要安装几个包：

```
$ pip install virtualenv
$ pip install virtualenvwrapper
```

第一个包是创建虚拟环境的基础，详情参见 Jamie Matthews 写的这篇文章。然而，virtualenv 包用起来不是很顺手。第二个包包装了 virtualenv 包的功能，省时省力。

如果你使用的是基于 Linux/Unix 的操作系统，为了使用 virtualenvwrapper，只需执行一个 shell 脚本：

```
$ source virtualenvwrapper.sh
```

通常，我们会把这一行代码添加到 bash/profile 脚本中，以免每次想使用虚拟环境时都自己动手运行。

Windows 用户要安装 virtualenvwrapper-win 包：

```
$ pip install virtualenvwrapper-win
```

接下来可以创建虚拟环境了：

```
$ mkvirtualenv rango
```

若想查看有哪些虚拟环境，使用 `lsvirtualenv` 命令。下述命令激活一个虚拟环境：

```
$ workon rango
(rango) $
```

激活虚拟环境后，提示符中会显示当前虚拟环境的名称，即这里的 `rango`。现在，你可以放心在这个环境中安装所需的包，无需担心会对其他环境产生影响。执行 `pip list` 命令，看看你的虚拟环境中有没有安装 Django 和 Pillow。如果没有，使用 pip 把它们安装到你的虚拟环境中。

A.5 版本控制

我们建议，代码应该使用版本控制工具管理，例如 SVN 或 Git。如果你没用过 Git 和 GitHub，建议你阅读《GitHub入门与实践》一书。我们建议你把自己的项目纳入 Git 仓库，这样能避免很多灾难。

B 中期练习参考解答

希望第 14 章给出的步骤能协助你完成练习。如果你做不出来，需要一点提示，请参照本篇附录给出的方法。

> **有更好的方法？**
>
> 这里给出的方法只是解决问题的方式之一。如果你以其他方式实现了，请与笔者分享你的经验。还可以在 Twitter 上 @tangowithdjango，让更多的人知道。

B.1 记录网页的访问次数

编写视图

在 */rango/views.py* 模块中定义一个新视图，命名为 `track_url()`。这个视图从 HTTP GET 请求参数（*rango/goto/?page_id=1*）中提出所需的信息，更新相应网页的访问次数，然后重定向到真正的 URL。

```
from django.shortcuts import redirect

def track_url(request):
    page_id = None
    url = '/rango/'
    if request.method == 'GET':
        if 'page_id' in request.GET:
            page_id = request.GET['page_id']

            try:
```

```
                page = Page.objects.get(id=page_id)
                page.views = page.views + 1
                page.save()
                url = page.url
            except:
                pass

    return redirect(url)
```

别忘了在 *views.py* 模块顶部导入 `redirect()` 函数。

```
from django.shortcuts import redirect
```

添加 URL 映射

打开 */rango/urls.py* 模块，把下述代码添加到 `urlpatterns` 中。

```
url(r'^goto/$', views.track_url, name='goto'),
```

修改分类页面的模板

修改 *category.html* 模板，不让用户点击真正的 URL，而是使用 *rango/goto/?page_id=XXX*。

```
{% for page in pages %}
    <li>
        <a href="{% url 'goto' %}?page_id={{page.id}}">{{ page.title }}</a>
        {% if page.views > 1 %}
            ({{ page.views }} views)
        {% elif page.views == 1 %}
            ({{ page.views }} view)
        {% endif %}
    </li>
{% endfor %}
```

可以看到，我们根据 `page.views` 的值显示“view”、“views”，或者什么也不显示。

修改分类视图

既然已经记录了网页的访问次数，那么可以更改 `category()` 视图，按访问次数排序网页：

```
pages = Page.objects.filter(category=category).order_by('-views')
```

点击几个链接，然后返回分类页面，确认一切正常。再看看其他分类页面。

B.2 在分类页面中搜索

去掉搜索页面

首先要删除独立的搜索页面，只让用户在分类页面中搜索。为此，要删除现在的搜索页面及其视图。

然后，修改 *base.html* 模板，把导航栏中的“Search”链接删掉。*rango/urls.py* 模块中的 URL 映射也要删除。

添加搜索表单

打开 *category.html* 模板，在网页列表下方添加搜索表单。这与 *search.html* 模板中的代码差不多，但是要把 `action` 属性指向 `show_category` 页面。此外还要传入 `query` 变量，让用户知道自己搜索的是什么。

```
<form class="form-inline" id="user_form"
    method="post" action="{% url 'show_category'  category.slug %}">
    {% csrf_token %}
    <div class="form-group">
        <input class="form-control" type="text" size="50"
            name="query" value="{{ query }}" id="query" />
    </div>
    <button class="btn btn-primary" type="submit" name="submit"
        value="Search">Search</button>
</form>
```

表单下方显示搜索结果。同样，模板代码也与 *search.html* 模板中的类似。

```
<div>
{% if result_list %}
    <h3>Results</h3>
    <!-- 按顺序显示搜索结果 -->
    <div class="list-group">
```

```
        {% for result in result_list %}
            <div class="list-group-item">
                <h4 class="list-group-item-heading">
                    <a href="{{ result.link }}">{{ result.title }}</a>
                </h4>
                <p class="list-group-item-text">{{ result.summary }}</p>
            </div>
        {% endfor %}
        </div>
    {% endif %}
    </div>
```

别忘了把搜索表单和搜索结果放在 `{% if user.authenticated %}` 和 `{% endif %}` 之间，只让通过身份验证的用户搜索，以免访客浪费有限的 API 使用额度。

修改分类视图

修改 `show_category()` 视图，处理 HTTP POST 请求，并把搜索结果列表传给模板上下文。

```
def show_category(request, category_name_slug):
    # 创建上下文字典，稍后传给模板渲染引擎
    context_dict = {}

    try:
        # 能通过传入的分类别名找到对应的分类吗?
        # 如果找不到，.get() 方法抛出 DoesNotExist 异常
        # 因此 .get() 方法返回一个模型实例或抛出异常
        category = Category.objects.get(slug=category_name_slug)
        # 检索关联的所有网页
        # 注意，filter() 返回一个网页对象列表或空列表
        pages = Page.objects.filter(category=category)

        # 把得到的列表赋值给模板上下文中名为 pages 的键
        context_dict['pages'] = pages
        # 也把从数据库中获取的 category 对象添加到上下文字典中
        # 我们将在模板中通过这个变量确认分类是否存在
        context_dict['category'] = category
    except Category.DoesNotExist:
        # 没找到指定的分类时执行这里
```

```python
            # 什么也不做
            # 模板会显示消息，指明分类不存在
            context_dict['category'] = None
            context_dict['pages'] = None


        # 新添加的处理 POST 请求的代码

        # 设定一个默认的搜索词条（分类名），显示在搜索框中
        context_dict['query'] = category.name

        result_list = []
        if request.method == 'POST':
            query = request.POST['query'].strip()

            if query:
                # 向搜索 API 发起请求，获得结果列表
                result_list = run_query(query)
                context_dict['query'] = query
                context_dict['result_list'] = result_list


        # 渲染响应，返回给客户端
        return render(request, 'rango/category.html', context_dict)
```

注意，context_dict 现在多了 result_list 和 query。如果没有搜索词条，使用默认的值，即分类名。这个值在搜索框中显示。

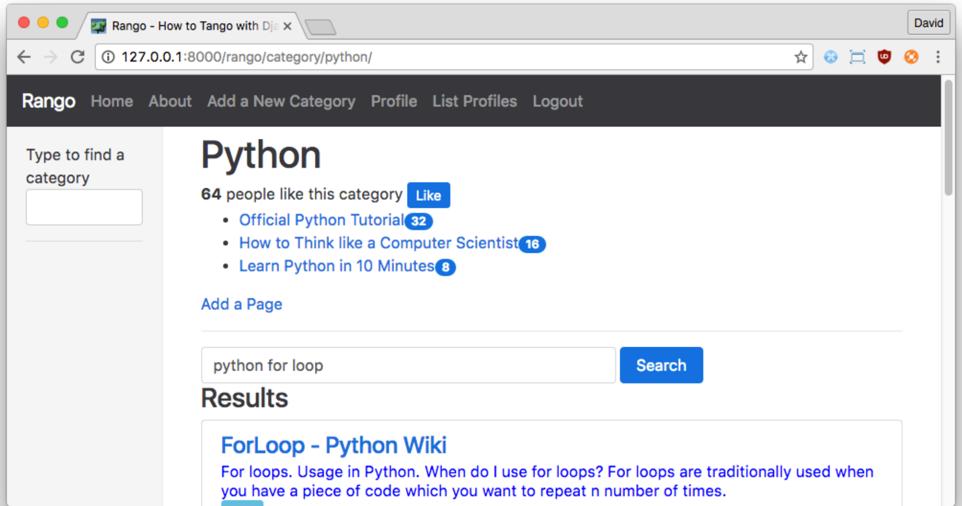


图 B-1：添加搜索功能后的分类页面

B.3 增加个人资料页面

本节为 Rango 应用的用户增加个人资料页面。Django 自带的 `User` 对象包含一些关于用户的基本信息，例如用户名和密码。但是我们实现了 `UserProfile` 模型，存储一些额外信息，例如用户网站和头像。基本步骤如下：

- 创建 *profile_registration.html* 模板，显示 `UserProfileForm`。
- 定义 `UserProfileFormModelForm` 类，处理这个新表单。
- 编写 `register_profile()` 视图，捕获用户提供的个人信息。
- 把视图映射到 URL *rango/register_profile/* 上。
- 修改 `MyRegistrationView` 类中的 `get_success_url()` 方法，把重定向地址改为 *rango/add_profile/*。

注册新用户的步骤如下：

- 点击“Register”链接。
- 在表单中填写基本的信息。
- 在新增的 `UserProfileForm` 表单中填写额外信息。
- 完成注册。

这个过程假设保存个人资料表单之前已经注册了用户。

创建模板

首先创建一个模板，显示用于填写额外信息的注册表单。这里，为了便于说明，我们特意把 django-registration-redux 表单与新增的 `UserProfileForm` 表单分开。如果你想合并二者，为什么不试试呢?

在 Rango 应用的模板目录中新建一个模板文件，命名为 *profile_registration.html*，写入下述 HTML 标记和 Django 模板代码。

```
{% extends "rango/base.html" %}

{% block title_block %}
    Registration - Step 2
{% endblock %}

{% block body_block %}
    <h1>Registration - Step 2</h1>
    <form method="post" action="." enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Submit" />
    </form>
{% endblock %}
```

这与现在的登录表单很像，也继承自 *base.html* 模板，共用基本的页面结构。我们在 `body_block` 区块中添加了一个 HTML 表单，使用对应视图传入模板的 `form` 对象生成字段。

◆ 别忘了 `multipart/form-data` ◆

别忘了在 `<form>` 标签中设定 `enctype="multipart/form-data"` 属性，以此告诉 Web 浏览器和服务器，不要编码任何数据，因为这个表单要处理文件上传。如若不然，这个表单将无法上传头像。

定义 UserProfileForm 类

Rango 应用的 *models.py* 模块中有个 UserProfile 模型，下面列出它的代码，以防你忘了。

```
class UserProfile(models.Model):
    # 这一行是必须的
    # 建立与 User 模型之间的关系
    user = models.OneToOneField(User)

    # 想增加的属性
    website = models.URLField(blank=True)
    picture = models.ImageField(upload_to='profile_images', blank=True)

    # 覆盖 __str__() 方法，返回有意义的字符串
    # 如果使用 Python 2.7.x，还要定义 __unicode__ 方法
    def __str__(self):
            return self.user.username
```

为了能够自动生成对应表单的 HTML 标记，我们要根据 UserProfile 模型定义一个 ModelForm 的子类。根据第 7 章所学的知识，可以像下面这样定义 UserProfileForm 类。

```
class UserProfileForm(forms.ModelForm):
    website = forms.URLField(required=False)
    picture = forms.ImageField(required=False)

    class Meta:
        model = UserProfile
        exclude = ('user',)
```

注意，website 和 picture 字段是可选的（required=False）。我们在嵌套的 Meta 类中指定，这个表单对应的模型是 UserProfile。exclude 属性告诉 Django，不要为 user 模型属性生成表单字段。因为新注册的用户还没有 User 对象可引用，所以后面我们将手动关联到 UserProfile 实例上。

编写视图

接下来要编写一个视图，处理 UserProfileForm 表单，新建 UserProfile 实例，然后使用 *profile_registration.html* 模板渲染响应。现在，你应该知道如何编写这样的视图了。这个表单既要能处理 HTTP GET 请求（渲染表单），还要能处理 HTTP POST 请求（处理输入的新信息）。下面

是一种可行的实现方式。

```
@login_required
def register_profile(request):
    form = UserProfileForm()

    if request.method == 'POST':
        form = UserProfileForm(request.POST, request.FILES)
        if form.is_valid():
            user_profile = form.save(commit=False)
            user_profile.user = request.user
            user_profile.save()

            return redirect('index')
        else:
            print(form.errors)

    context_dict = {'form':form}

    return render(request, 'rango/profile_registration.html', context_dict)
```

创建 `UserProfileForm` 实例之后，通过 `request` 对象判断是 GET 请求还是 POST 请求。如果是 POST 请求，使用 POST 请求中的数据重新创建 `UserProfileForm` 实例。因为还要处理图像上传（用户的头像），因此我们通过 `request.FILES` 获取上传的文件。随后检查提交的表单数据是否有效，即判断有没有正确填写表单字段。这里只是检查填写的 URL 是否有效，因为两个字段都是可选的。

如果提交的数据有效，创建 `UserProfile` 实例：`user_profile = form.save(commit=False)`。我们指定了 `commit=False` 参数，以便在保存到数据库中之前做些修改。我们要做的是把 `UserProfile` 实例与前面创建的 `User` 对象关联起来。成功保存 `UserProfile` 实例后，重定向到 `index` 视图。如果出于什么原因，提交的表单无效，在控制台中打印错误。在真实的代码中，你可能会想以更好的方式处理错误。

如果是 HTTP GET 请求，说明只是想要一个空表单，供用户填写。因此我们使用空的 `UserProfileForm` 实例（模板上下文中的 `form`）渲染 *profile_registration.html* 模板。

如此一来，这个视图既能显示 `UserProfileForm` 表单，又能解析并保存表单中填写的数据。

◆ 找不到 login_required? ◆

注意，新注册的用户访问这个视图时已经有账户了，因此可以放心地假设用户已登录。因此我们才使用 `@login_required` 装饰器，以防未经授权的人访问这个视图。

如果报错找不到 `login_required()` 函数（用作装饰器），确保 *views.py* 模块的顶部有下述 `import` 语句。

```
from django.contrib.auth.decorators import login_required
```

映射 URL

模板和视图准备好之后，有经验的 Django 开发者顺其自然会想到，接下来应该映射 URL 了。我们要把视图映射到一个 URL 上，让用户能访问刚创建的内容。打开 Rango 应用的 *urls.py* 模块，把下面这行代码添加到 `urlpatterns` 列表中。

```
url(r'^register_profile/$', views.register_profile, name='register_profile'),
```

这行代码把新增的 `register_profile()` 视图映射到 URL */rango/register_profile/* 上。还记得吗，URL 中的 */rango/* 部分来自项目的 *urls.py* 模块，余下的部分则由 Rango 应用的 *urls.py* 模块负责。

调整注册流程

至此，一切（基本）就绪了，接下来要调整用户的注册流程。前面，我们定义了一个基于类的视图，名为 `MyRegistrationView`。在那个类中我们修改了成功注册后的重定向地址，转到 Rango 应用的首页。现在，我们要把重定向地址改为这个填写额外信息的页面。前一小结把这个页面的 URL 命名为 `register_profile`，因此要像下面这样修改 `MyRegistrationView` 类。

```
class MyRegistrationView(RegistrationView):
    def get_success_url(self, user):
        return reverse('register_profile')
```

现在，注册账户时，先重定向到这个页面填写额外信息，然后再重定向到 Rango 应用的首页。

■ 基于类的视图 ■

这一小节提到了“基于类的视图”。基于类的视图与基于函数的视图有所不同，它更优雅，但是也更复杂。本书一直使用基于函数的视图，我们在 *views.py* 模块中定义了一些视图函数，

处理不同的请求。基于类的视图利用继承，通过实现一系列方法处理请求。例如，在视图函数中我们时常会判断是 GET 请求还是 POST 请求，但是在基于类的视图中，我们要分别实现 `get()` 和 `post()` 方法。如果项目比较复杂，要处理的情况较多，使用基于类的视图更合适。详情参见 Django 文档。

</> 附加练习

- 阅读 Django 文档，学习如何编写基于类的视图。
- 把 Rango 应用中的视图函数改成基于类的视图。
- 在 Twitter 中发表你的感受，并 @tangowithdjango。

B.4 查看个人资料

有了个人资料，我们还要让用户能查看和编辑自己的个人资料。实现的过程也跟前面差不多。基本步骤如下：

- 新建一个模板，命名为 *profile.html*。
- 编写一个视图，命名为 `profile()`，渲染 *profile.html* 模板。
- 把 `profile()` 视图映射到一个 URL 上（*/rango/profile*）。

此外，还要在 Rango 应用的 *base.html* 模板中添加一个链接，指向这个新视图。我们将创建一个通用的视图，能查看任何用户的信息。已登录的用户还可以编辑个人资料，但只能编辑自己的。

创建模板

首先创建一个简单的模板，显示用户的个人资料。在 Rango 应用的模板目录中新建 *profile.html* 模板文件，写入下述 HTML 标记和 Django 模板代码。

```
{% extends 'rango/base.html' %}

{% load staticfiles %}

{% block title %}{{ selecteduser.username }} Profile{% endblock %}
```

```
{% block body_block %}

<h1>{{selecteduser.username}} Profile</h1>
<img src="{{ MEDIA_URL }}{{userprofile.picture }}"
     width="300"
     height="300"
     alt="{{selecteduser.username}}" />
<br/>
<div>
    {% if selecteduser.username == user.username %}
        <form method="post" action="." enctype="multipart/form-data">
             {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Update" />
        </form>
    {% else %}
    <p><strong>Website:</strong> <a href="{{userprofile.website}}">
        {{userprofile.website}}</a></p>
    {% endif %}
</div>
<div id="edit_profile"></div>
{% endblock %}
```

注意，我们要在模板上下文中定义几个变量：selecteduser、userprofile 和 form。

这个模板的重点在 body_block 区块中，首先显示用户的头像，然后通过 form 变量生成一个表单，让用户修改自己的信息。这个表单仅在选择的用户与当前登录的用户一致时才显示，也就是只让登录的用户编辑自己的个人资料。如果选择的用户与当前登录的用户不一致，那就只显示用户的个人资料，但是不能编辑。

此外请注意，这个表单也设定了 enctype="multipart/form-data"，因为要上传图像。

编写视图

根据上述模板，下面编写一个简单的视图，处理对用户个人资料的查看和表单数据的提交。在 Rango 应用的 *views.py* 模块中，定义一个名为 profile() 的视图。

```
@login_required
def profile(request, username):
```

```
    try:
        user = User.objects.get(username=username)
    except User.DoesNotExist:
        return redirect('index')

    userprofile = UserProfile.objects.get_or_create(user=user)[0]
    form = UserProfileForm(
        {'website': userprofile.website, 'picture': userprofile.picture})

    if request.method == 'POST':
        form = UserProfileForm(request.POST, request.FILES, instance=userprofile)
        if form.is_valid():
            form.save(commit=True)
            return redirect('profile', user.username)
        else:
            print(form.errors)

    return render(request, 'rango/profile.html',
        {'userprofile': userprofile, 'selecteduser': user, 'form': form})
```

这个视图要求用户已登录，因此我们加上了 `@login_required` 装饰器。首先，从数据库中查询指定的 `django.contrib.auth.Use` 对象。如果找不到用户，重定向到 Rango 应用的首页，而不显示错误消息——不给不存在的用户提供任何信息。如果用户存在，再获取相应的 `UserProfile` 实例。如果用户还没设定个人资料，可以创建一个空的 `UserProfile` 对象。随后创建一个 `UserProfileForm` 对象，使用找到的用户的个人信息填充。这个表单是否呈现给用户由模板控制。

接下来判断是不是 HTTP POST 请求，即是不是提交表单更新个人资料。如果是，从请求数据中提取信息，新建一个 `UserProfileForm` 实例。这里我们把 `instance` 参数设为前面创建的 `UserProfile` 模型实例，而不每次都新建。注意，这是更新，而不是新建。如果表单数据有效，保存。如果数据无效，或者是 HTTP GET 请求，渲染 *profile.html* 模板。

</> 一道简单的练习

如何修改上述代码，以免用户修改别的用户的个人资料？为此要添加什么条件判断？

映射 URL

接下来要把 `profile()` 视图映射到一个 URL 上。跟之前一样，我们要在 Rango 应用的 *urls.py* 模块中添加一行代码。把下述代码添加到 `urlpatterns` 列表的末尾：

```
url(r'^profile/(?P<username>[\w\-]+)/$', views.profile, name='profile'),
```

注意，正则表达式中有个 `username` 变量，它匹配 */profile/* 之后的内容。例如，对 URL */rango/profile/maxwelld90* 来说，`username` 的值是 `maxwelld90`，这也就是传给 `profile()` 视图的 `username` 参数的值。我们就是这样判断当前查看的是哪个用户的个人资料。

调整基模板

现在一切应该都能按预期使用了，不过最好在 Rango 应用的 *base.html* 模板中添加一个链接，指向当前登录用户的个人资料页面，方面用户查看和编辑自己的个人资料。打开 Rango 应用的 *base.html* 模板，找到导航栏中为已登录用户显示的那些链接。在其中添加下述链接：

```
<a href="{% url 'profile' user.username %}">Profile</a>
```

你还可以为这个链接添加额外信息，例如为 `<a>` 标签添加 `class` 属性，以便添加样式。

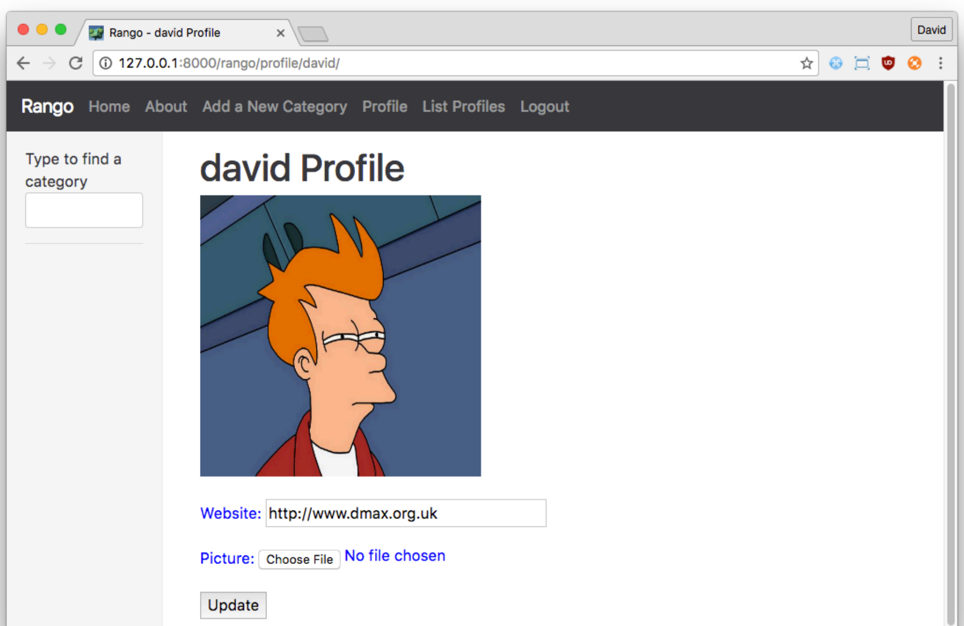


图 B-2：用户个人资料页面

B.5 列出所有用户

最后一项挑战是再创建一个页面，列出 Rango 应用的所有用户。这个任务不难，我们需要创建一个模板、编写一个视图，再映射 URL。我们需要获取在 Rango 应用中注册的所有用户，并为每个用户添加一个链接，以便查看他们的个人资料。

创建模板

在 Rango 应用的模板目录中，创建一个名为 *list_profiles.html* 的模板，写入下述 HTML 标记和 Django 模板代码。

```
{% extends 'rango/base_bootstrap.html' %}

{% load staticfiles %}

{% block title %}User Profiles{% endblock %}

{% block body_block %}
<h1>User Profiles</h1>

<div class="panel">
    {% if userprofile_list %}
    <div class="panel-heading">
        <!-- 按顺序显示搜索结果 -->
        <div class="panel-body">
            <div class="list-group">
                {% for listuser in userprofile_list %}
                <div class="list-group-item">
                    <h4 class="list-group-item-heading">
                        <a href="{% url 'profile' listuser.user.username %}">
                            {{ listuser.user.username }}</a>
                    </h4>
                </div>
                {% endfor %}
            </div>
        </div>
    </div>
    {% else %}
        <p>There are no users for the site.</p>
```

```
        {% endif %}
    </div>
    {% endblock %}
```

这个模板没什么难理解的，我们创建了一系列 `<div>` 标签，并设定了 `class` 属性，使用 Bootstrap 装饰。对找到的每个用户，显示其用户名，并提供一个链接，指向那个用户的个人资料页面。注意，我们迭代的是 `UserProfile` 对象列表，因此要通过 `UserProfile` 对象的 `user` 属性获取 `username`。

编写视图

接下来要编写对应的视图，从 `UserProfile` 模型中检索全部用户。我们假定只有已登录用户才能查看这个视图。把下述 `list_profiles()` 视图添加到 Rango 应用的 *views.py* 模块中。

```python
@login_required
def list_profiles(request):
    userprofile_list = UserProfile.objects.all()

    return render(request, 'rango/list_profiles.html',
        {'userprofile_list' : userprofile_list})
```

映射 URL，添加链接

最后，把 `list_profiles()` 视图映射到一个 URL 上。打开 Rango 应用的 *urls.py* 模块，把下述代码添加到 `urlpatterns` 列表里：

```python
url(r'^profiles/$', views.list_profiles, name='list_profiles'),
```

此外，还可以在 Rango 应用的 *base.html* 模板中添加一个链接，让已登录用户查看这个页面。跟前面一样，把下述标记添加到只有登录用户才能看到的那些链接当中。

```html
<a href="{% url 'list_profiles' %}">List Profiles</a>
```

添加这个链接后，你便可以查看用户列表和各用户的个人资料了。

</> 改进个人资料页面

- 修改用户列表，显示用户的头像。
- 如果用户没有上传头像，显示默认的图片。可以使用 LoremPixel 这种服务生成图像。

★ 提示 ★

可以使用 `<img width="64" height="64" src="http://lorempixel.com/64/64/people/"/>` 从 LoremPixel 获取一张 64x64 的头像。注意，可能要等几秒钟图片才能下载好。